Dynamic platform-independent meta-algorithms for graph-partition

Schwartz, Victor Scott

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DYNAMIC PLATFORM-INDEPENDENT META-ALGORITHMS FOR GRAPH-PARTITIONING

By

Victor S. Schwartz

September 1998

Thesis Advisor: Gordon H. Bradley
Second Reader: R. Kevin Wood

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1998	Master's Thesis

4. TITLE AND SUBTITLE
Dynamic Platform-Independent Meta-Algorithms for Graph-Partitioning

5. FUNDING NUMBERS
N0001498WR20001

6. AUTHOR(S)
Schwartz, Victor Scott

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Office of Scientific Research, 110 Duncan Avenue, Suite 100, Bolling AFB, DC-0001

Office of Naval Research, 800 North Quincy Street, Arlington, VA 22217

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

A dynamic platform-independent solver is developed for use with network and graph algorithms of operations research. This solver allows analysts to solve a large variety of problems without writing code. Algorithms from a library can be integrated into a meta-algorithm which also provides easy monitoring of solution progress.

The solver, DORS, is demonstrated by heuristically solving a graph-partitioning problem to minimize the number of nodes adjacent to other segments of the partition. The model arises from a network-upgrade project faced by the Defense Information Systems Agency (DISA), a problem with over 200 nodes and 1400 arcs. Solutions are provided on a 266 MHz Pentium II PC using Windows NT 4.0. Eight variants of the problem are solved involving modification to the objective function, constraints on the size of partition segments, and on the number of those segments.

DORS (and the meta-algorithm it implements) appears to find a good solution for one of the two problem formulations for DISA, but has difficulty solving the other. Because the solver allows new algorithms to be easily added to create more powerful meta-algorithms, DORS should provide a good solution approach for both problem formulations given a more versatile library of algorithms.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Graph Partitioning, Java	118
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev.2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited

DYNAMIC PLATFORM-INDEPENDENT META-ALGORITHMS FOR GRAPH-PARTITIONING

Victor Scott Schwartz
Lieutenant, United States Navy
B.S., University of Nebraska 1992

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL
September 1998

ABSTRACT

A dynamic platform-independent solver is developed for use with network and graph algorithms of operations research. This solver allows analysts to solve a large variety of problems without writing code. Algorithms from a library can be integrated into a meta-algorithm which also provides easy monitoring of solution progress.

The solver, DORS, is demonstrated by heuristically solving a graph-partitioning problem to minimize the number of nodes adjacent to other segments of the partition. The model arises from a network-upgrade project faced by the Defense Information Systems Agency (DISA), a problem with over 200 nodes and 1400 arcs. Solutions are provided on a 266 MHz Pentium II PC using Windows NT 4.0. Eight variants of the problem are solved involving modification to the objective function, constraints on the size of partition segments, and on the number of those segments.

DORS (and the meta-algorithm it implements) appears to find a good solution for one of the two problem formulations for DISA, but has difficulty solving the other. Because the solver allows new algorithms to be easily added to create more powerful meta-algorithms, DORS should provide a good solution approach for both problem formulations given a more versatile library of algorithms.

THESIS DISCLAIMER

The reader is cautioned that the computer program developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the program is free of computational and logic errors, it cannot be considered validated. Any application of this program without additional verification is at the risk of the user.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

EXECUTIVE SUMMARY

This thesis develops "DORS," a dynamic platform-independent solver for use with network and graph algorithms of operations research (OR). DORS will eventually allow analysts to solve a large variety of problems without writing code: Simple algorithms from a library can be combined within DORS into a powerful meta-algorithm. DORS also provides easy monitoring of solution progress.

DORS is demonstrated on a graph-partitioning problem designed to assist the Defense Information Systems Agency (DISA) in upgrading the Defense Information Infrastructure (DII). This problem is heuristically solved by combining a library of algorithms and a dynamic interface. The interface is provided through proper object management and a graphical user interface.

Graph partitioning is widely used in OR and computer science. It is the problem of partitioning the nodes of an undirected graph into subsets of a specified size and/or number so that the sum of the arc weights (or simply the number of arcs) crossing between segments is minimized. Other objective functions, like those used here, can be modeled too, e.g., the total number of nodes adjacent to nodes in other segments.

DISA wishes to partition a portion of the DII - the portion is a network with over 200 nodes and 1400 arcs - into four or five segments and upgrade the hardware in one segment at a time. DISA's problem can be viewed as a variant of graph partitioning.

Many graph-partitioning algorithms have been developed, but they are usually re-written for each problem to which they are applied. There is a need for a computer program and a library of algorithms that can be easily accessed and applied to a variety of graph-partitioning problems without modifying code. DORS fills this need.

DORS (and the meta-algorithm it implements) appears to find a good solution for one of the two problem formulations for DISA, but has difficulty solving the other. Because the solver allows new algorithms to be easily added to create more powerful meta-algorithms, DORS should provide a good solution approach for both problem formulations given a more versatile library of algorithms. (Computation is performed on a 266 MHz Pentium II PC operating under Windows NT 4.0.)

As new algorithms are added to DORS and monitoring methods developed for it, DORS will become a powerful, standard tool for solving OR problems.

I. INTRODUCTION

The Defense Information Infrastructure (DII), which is the network of data-transmission facilities maintained by the Defense Information Systems Agency (DISA), is outdated and must be upgraded to newer and faster technologies. The proposed upgrade will not only increase system speed but will also allow for easy modifications in reaction to future increases in user demand.

For the portion of this system in the continental United States, DISA intends to upgrade the system by partitioning the system into four or five approximately equal-sized pieces, and then upgrade the hardware in one segment of the partition at a time. One key contribution of this thesis is the modeling of the problem in terms of graph partitioning. We refer to the proposed graph-partitioning models or "problems" as the DISA Transmission Facilities Improvement Project (DTFIP). (See references [14] and [17] for basic background in graph partitioning.) DISA's goal is to maintain connectivity of the DII throughout the project in order to minimize the impact on their customers. An important secondary concern is to minimize the costs of the upgrade.

Two distinct graph-partitioning problems arise out of DTFIP and its cost and connecting concerns. The second key contribution of this thesis is the development of a solver for these problems, a solver that allows an analyst to construct dynamic platform-independent meta-algorithms for the problems.

A. BACKGROUND

DISA maintains worldwide transmissions facilities in support of the Department of Defense (DoD). However, the first stage of DTFIP, and the scope of this thesis, is limited to the continental United States. The section of the DII facilities discussed in this thesis is a network consisting of roughly 200 switching stations (nodes in the transmission network) and roughly 1400 connecting communications lines (arcs in the network).

DISA's primary mission is "to plan, engineer, develop, test, manage programs, acquire, implement, operate, and maintain information systems for Command, Control, Computers, Communications, and Information (C4I) and mission support under all conditions of peace and war" [4]. DII is the backbone of the DoD's C4I network. It provides

the connections and interfaces that makes DISA-managed programs a system capable of sharing data and resources.

A degradation of the DII would adversely affect DISA's four primary mission areas, namely, the Global Command and Control System (GCCS), Defense Messaging System (DMS), Defense Information System Network (DISN), and Global Combat Support System (GCSS). These branches of DII provide the foundation for national defense assets by integrating all military command, control, and information through dispersed UNIX and personal computers tied together through the Secret Internet Protocol Router Network (SIPRNET) [4]. SIPRNET is an encrypted extension of the DII.

GCCS is a system of databases and applications that provides support to the nation's war-fighters. The primary use of GCCS is C4I: GCSS provides the tie between the intelligence-collection agencies, command staffs, and unit commanders in the field. These customers rely on the information passing through DISA's network in support of GCCS to keep them apprised of all aspects of warfare. GCCS provides real-time data critical in countering enemy actions and protecting US assets and personnel. An

interruption to this system would almost bring the DoD to a halt until alternate systems were brought on line.

The DII currently operates using Statistical Multiplexing. This technology allows a data transfer speed of 19.2 kilobits per second (kBps) per channel and aggregate transfer speeds of up to 76.8 kBps when multiplexing eight channels [7]. These speeds were the best available when the network was built, but are becoming more and more inadequate as the databases being accessed through the DII become larger and DoD intelligence traffic increases. In order to keep up with technology and improve customer service, DISA plans to upgrade their switches to Asynchronous Transmission Mode (ATM).

ATM switches allow for basic data transmission rates in excess of 155 Megabits per second (MBps) with the ability to easily increase the rates through improved multiplexing and the use of multiple transmission lines [6]. In addition to the orders-of-magnitude increase in transfer rates, ATM switches allow improvement in transmission speeds as technology advances through better multiplexing and compression routines. The traditional limit of eight lines for multi-channel transmissions is also broken.

Because the DII is so vital, its functionality must not be significantly degraded during the upgrade; the system must maintain connectivity throughout this process. In order to keep service interruptions to an absolute minimum, any switch that communicates with both old and new switches must maintain the older Statistical Multiplexing and the new ATM switching circuitry. When all connected facilities have been upgraded and the Statistical Multiplexing switches are no longer needed, they may be removed.

B. META-ALGORITHMS

DTFIP, the DISA upgrade problem, may be modeled using variants of the classic graph-partitioning problem (e.g., [3], [14], and [15]). The basic problem of DTFIP is to minimize the number of "interface nodes" that are connected to nodes in other segments of the partition and thus require interface hardware. But, different upgrade models lead to two objective functions: The first objective minimizes the total number of interface nodes, and the second minimizes the number of interface nodes needed at any one time. These objectives are explained in further detail in Chapter III.

Graph-partitioning problems are usually solved using heuristics. A heuristic algorithm is a method of searching for an optimal solution to a problem; usually it will construct a good solution but cannot guarantee that an optimal solution will be found.

Graph-partitioning problems may also be solved using integer-programming techniques (e.g.,[17]). Solution times for integer programs tend to grow exponentially in problem size, and therefore a solution may not be produced in a timely manner. A literature review indicates that heuristic algorithms are generally accepted as a better approach for handling large graph-partitioning problems.

There are a variety of heuristic algorithms that may be applied to graph partitioning; each has its advantages. The quickest heuristic is a greedy algorithm that operates on each node only once. The algorithm assigns each node to the best available partition segment; this choice is based on the current solution characteristics of the node being assigned, the change in a partial or complete objective function, etc. Although a feasible solution is found very quickly, the algorithm is myopic and the solution may be far from optimal [14].

Commonly, an initial solution is obtained with a greedy algorithm and this solution is then refined with a local search [9] which finds a local optimum "near" the initial solution. Once a local optimum is found, a more complicated search technique may be used to construct an improved solution that is locally optimal with respect to the more complicated search.

One local search technique simply enumerates every possible solution in the vicinity of the current solution. This method may be used to find a local optimum using an initial solution and is guaranteed to find a better solution if one exists in its search area. This method can be focused on a very narrow section of the problem space or it may be carried to the extreme of enumerating every possible solution. The key to such an algorithm is determining the right tradeoff between increased enumeration and improved solutions [15].

An alternative to exhaustive searches over a subset of the problem space is to "shock" the solution through drastic changes in order to force the algorithm out of a local optimum. Such a shock moves some subset of the nodes into different partition segments. After the shock, the solver can switch back to a local search that will find a

local optimum. Since the changes produced by the shock are random, they are not guaranteed to find a better solution, even if one exists, unless they are run indefinitely. Since we are time-constrained, this means that we may not find the optimal solution.

Shocks change part of the solution while leaving part of it intact. The idea is to escape a local optimum through a random move while retaining part of the best solution found so far. Once a new solution is found, a deterministic technique may be used to find a local optimum in the vicinity of the shocked solution. It is hoped that the new local optimum will be an improvement over the previous best solution. It is difficult to determine the size and frequency with which the shocks should be used, but commonly, such an algorithm starts with large shocks that are slowly decreased in magnitude [15].

There are many algorithms for graph-partitioning not discussed in this thesis. For example, Recursive Orthogonal Bisection [8], Spectral Multi-section [2], Neural Networks [10], Genetic Algorithms [11], Mean Field Annealing [12], and Simulated Annealing [12] have all been used effectively. The variety of heuristic algorithms available for graph partitioning and the large number of

factors that may be fine-tuned leads to the concept of a meta-algorithm.

It is difficult to construct good solutions to graph-partitioning problems. Algorithms for this problem often quickly determine a reasonably good solution but then consume significant computer time to produce little or no improvement. Often, a sequence of algorithms, each executed for a short time, finds a better solution than running one algorithm for a long time.

A sequence of algorithms is called a "meta-algorithm." Specifying an appropriate sequence of algorithms is a difficult task because a meta-algorithm that works well on one class of problems may not work well on another class. Thus, an analyst faced with solving a specific problem, such as the DTFIP, may want to try several meta-algorithms to determine which is best for the problem at hand.

This thesis develops a solver that allows an analyst to easily construct meta-algorithms that can then be used to solve the DTFIP graph-partitioning problems as well as other graph-partitioning problems. A solver to construct meta-algorithms should be dynamic, extensible, platform-independent, and capable of passive external monitoring.

These concepts will be explained in the following few pages.

1. Dynamic Algorithms

The solver used to form a meta-algorithm to solve DISA's problem, and other graph-partitioning problems, should be dynamic. A dynamic solver is one that allows an analyst to choose among a variety of model formulations and combination of algorithms to solve those formulations. The analyst is able to quickly change algorithms and compare results, and can exploit algorithms that he or she may be unable to code personally. This versatility should reduce the analyst's workload.

Often, the objective function of a problem is not well defined. The analyst must determine the customer's needs based on imperfect inputs. This uncertainty is further increased when a customer receives an initial solution to the problem at hand. The customer may see flaws in this solution and may then modify requirements. A dynamic solver will allow the analyst to compose a new meta-algorithm and change the objective-function evaluator quickly in response to these developments.

2. Extensible

The solver must be extensible. It should allow for the addition of new algorithms to its library and allow modification of existing algorithms without changing or recompiling the solver program. When the library is modified, the code for algorithms other than the one being added or modified is unaffected. Similarly, the solver should allow additional objective-function evaluators to be added or modified.

3. Platform-Independent Algorithms

Most computer programs can be executed in only one hardware/software environment. The term "platform-independent" refers to a program that can be executed without modification on a variety of hardware and software environments. For the sake of versatility, the solver program used in DTFIP should be platform-independent.

Analysts throughout the world work on a variety of computers. In order to facilitate analysts' use and to allow the analysts to access algorithms developed by others, contemporary solvers should be platform-independent. If platform-independent design isn't used,

isolation of algorithms to specific systems will continue unnecessary duplication of efforts.

4. External Passive Monitoring of Algorithms

The solver should allow monitoring of the progress of algorithms and monitoring of changes to the status of any property of the problem being analyzed. This monitoring should be controllable by the analyst, allowing a variety of monitoring methods and methods to display solver progress.

If monitoring is built into an algorithm, information that will be available to the analyst is predetermined. This thesis focuses on reusable code and methods to solve DTFIP. Since the needs of future analysts are not known, monitoring is best done by a method external to the algorithm to allow maximum versatility.

With passive external monitoring, there is only a small decrease in algorithmic performance to establish an interface that allows communication with "listener methods"[13]. Listener methods monitor algorithmic performance externally. They keep track of key changes to the solution as the algorithm runs and make this information available to other algorithms and programs.

Since changes to the solution are monitored through listener methods, the data extracted may be quickly modified to conform to the analyst's needs. For example, in the early stages of a meta-algorithm solution run, every change to the solution could be monitored and displayed. For our problem, this display could include a graph of the optimal solution and a color display of the network showing the best partition found. As the algorithm progresses and solution improvements are found less frequently, the analyst may decide to change the displays in use. For example, a graphical display of the solution could be displayed in the beginning while changes are rapid; and the analyst could switch to an algorithm status display when changes to the solution become infrequent.

Passive external monitoring also allows improved parallel processing. A listener module gathers data including solution improvements from the algorithm as it progresses, without interrupting the algorithm. This data is readily available to any method that wishes to monitor progress. If several computers run separate algorithms simultaneously, they can share the improvements, thus forming an interactive unit acting as a single processor.

The solver developed in this thesis is capable of passive external monitoring through the use of Konig[16]. However, the thesis does not demonstrate this capability.

5. Java-Based Technology

The requirements for a dynamic, extensible, platform-independent system with passive external monitoring can be satisfied if the system is written in the Java programming language [13]. Platform independence is fundamental to Java: The language is designed to run on any computer regardless of manufacturer and operating system. Java accomplishes this by generating architecture-neutral bytecode, i.e., low-level computer instructions that have nothing to do with a particular computer's architecture. The instructions are designed to be easy to interpret and translate into native machine code on the fly [13].

Java supports loading of classes at run-time and loading of classes while a program is executing. The solver is extensible in that algorithms can be added without modifying or recompiling the solver or the other algorithms. Algorithms are added to the solver dynamically after the solver has begun execution.

6. Konig for Graphs and Network Algorithms

Konig [16] is a software system for writing algorithms for graphs and networks. It can read and write graphs and networks and display their node and arc properties. Algorithms written in Konig can be passively monitored; the analyst can have access to activity on all nodes and arcs while the algorithm is executing. Konig is used to implement the algorithms in the solver.

C. STATEMENT OF THESIS

This thesis develops and demonstrates the Dynamic Operations Research Solver (DORS). It is a solver for dynamically constructing meta-algorithms for solving graph-optimization and network-optimization problems. In this thesis, DORS is limited to constructing meta-algorithms to solve graph-partitioning problems. However, its design is applicable to other operations research problems where dynamic, extensible meta-algorithms are likely to be beneficial. The DORS design provides a tool to solve operations research problems using a variety of algorithms and objective functions. Little programming is needed with DORS and sets of algorithms and objective-function evaluators may be changed easily. DORS allows the user to

spend time analyzing problems and possible solutions, rather than programming and debugging.

This thesis demonstrates the use of DORS to solve DTFIP. A meta-algorithm is composed using DORS for this purpose, a meta-algorithm that incorporates a variety of simple algorithms such as greedy and shocking heuristics.

This thesis consists of six chapters. Chapter II defines the DISA problem in detail and discusses assumptions. Chapter III defines key concepts used in the development of the solver and algorithms used to solve the problem. Chapter IV explains and demonstrates the use of DORS on the DISA problem. Chapter V summarizes the results and makes recommendations for upgrading DISA's transmission facilities. Finally, Chapter VI concludes the thesis.

II. DISA'S NETWORK-UPGRADE PROBLEM

This thesis investigates two possible solutions to DTFIP. DISA's objective with DTFIP is to minimize the cost of upgrading a portion of the DII while maintaining connectivity.

The portion of DII under study can be represented by a network G=(N,A) where N is a set of nodes and A is a set of undirected arcs (i,j) which are distinct, unordered pairs from N. The network nodes will be partitioned into K segments, $N_1, N_2, \ldots, N_K$. DISA expects K to be four or five. The nodes should be partitioned such that $|N_k| \approx |N|/K$, $k = 1, \ldots, K$. This thesis implements this requirement using constraints $m \leq |N_k| \leq M$ where $m = |N|/K - \delta$, $M = |N|/K + \delta$, $\delta > 0$. DISA would like $\delta \approx 10$ when $|N| \approx 200$. The values of K, m, M, and δ are specified by DISA and are subject to change at a future date. In the computational runs the values furnished by DISA are used.

In the first formulation, Problem 1, we wish to minimize the number of interface nodes N', i.e., nodes that are directly connected to one or more nodes in different partition segments so that interface hardware must be installed. The problem may be summarized as:

ObjectiveCountInterfaceNodes, Problem 1 Definition :
Given an undirected graph $G = (N, A)$ *with node set N and K segments,*
Find a K - partition of N, $\{N_1, N_2, \ldots, N_K\}$
Such that $m \le |N_k| \le M$ *for* $k = 1$ *to K, and*
So that $|N'|$ *is minimized, where*

$$N' = \bigcup_k \{i \in N_k \mid \exists (i, j) \in A \text{ with } j \notin N_k\}$$

In a second formulation of the DTFIP, Problem 2, we will allow the interface hardware of the interface nodes to be used multiple times. In this formulation, the partition segments are ordered from 1 to K, with the segments being upgraded in that order. The key issue is to reduce the maximum number of nodes required at any step of the upgrade. This second objective may be summarized as follows:

ObjectiveInterfaceNodesInOrder Problem2 Definition
Given an undirected graph $G = (N, A)$ *with node set N and K segments,*
Find an ordered K - partition of N, $\{N_1, N_2, \ldots, N_K\}$
Such that $m \le |N_k| \le M$ *for* $k = 1$ *to K and*
So that $|N''|$ *is minimized where*

$$N'' = \max_{k<K} \left\{ i \in \bigcup_{k'=1}^{k} N_{k'} \;\middle|\; \exists (i, j) \in A \text{ with } j \in \bigcup_{k'=k+1}^{K} N_{k'} \right\}$$

The first formulation minimizes the costs associated with installing and maintaining interface hardware by minimizing the number of interface nodes. The second formulation is likely to increase the number of interface

nodes and increase labor costs. As an offset, it reduces hardware costs associated with the upgrade. It is not clear which of the formulations will be more useful to DISA. Given both solutions, DISA should be able to compare potential labor costs from the first formulation with potential savings in hardware from the second and select the better solution.

III. DORS' KEY CONCEPTS

This chapter develops the key concepts associated with DORS and explains the major functions associated with the solver. It also provides detail on the operations of the specific functions used in the DTFIP. Chapter IV demonstrates these capabilities.

A. GRAPH ALGORITHMS

A generic graph algorithm to solve a generic problem takes a graph and possibly a candidate solution to the problem as input and performs its operations in order to obtain or improve a solution. This thesis demonstrates four algorithms to be used for graph partitioning namely, GetInitialSolution, Change1Node, Swap2Nodes, and RandomizeXNodes. GetInitialSolution creates a random starting solution; the other three algorithms operate on an input solution to improve it.

Algorithms in DORS are not tied to any specific objective function and do not know which objective function is used. Instead they access Evaluate_Objective which in turn forwards necessary information to the proper objective-function evaluator.

Before the meta-algorithm is executed, the analyst selects a particular objective function. Evaluate_Objective accesses this selection and ensures that the algorithm communicates with the set of objective-function evaluators associated with the objective function. The objective-function evaluator is responsible for the actual calculations; it is explained later in this chapter.

An algorithm in DORS maintains one local solution and its associated objective value. Algorithms, except for GetInitialSolution, have access to the best known solution and associated objective value found by previous algorithm calls. The solution maintained by the algorithm, the local solution, is "active." The active solution is being modified for potential improvements. The best solution is compared to the local solution and is updated if the local solution is an improvement.

Any algorithm may be assigned to perform its operations on a local solution from a previous algorithm call or on a copy of the best solution. In the latter case, a copy of the best solution becomes the local solution.

Algorithms in DORS access and store solutions related to the input graph using Konig [16], a Java-based language

that enables creation and control of graphs. Konig is also the mechanism through which graph-based information is stored and passive external monitoring is provided should an analyst develop code to monitor DORS' meta-algorithms.

1. GetInitialSolution

This algorithm provides an initial solution to the graph-partitioning problem by randomly assigning each node to a partition. It does not initially guarantee that this solution is feasible with respect to the cardinality constraints, $m \leq |N_k| \leq M$ for $k = 1,\ldots, K$. So after constructing a random solution, this algorithm calls MakeLegal, a segment of code that checks feasibility and, if necessary, modifies the solution to produce feasibility. MakeLegal is explained later in the chapter. After the solution is guaranteed to be feasible, the solution is sent to an objective-function evaluator to calculate the objective value.

A random solution is not the only way to obtain a starting solution. Simon [8] demonstrates that, in a planar graph, recursive coordinate bisection gives quick initial partitions while providing relatively good solutions. This algorithm is based on the fact that, in

planar graphs, nodes tend to be directly connected to nodes in close proximity. Therefore, proximate nodes will tend to be in the same partition.

It seems that the proximity argument might also apply to some non-planar graphs like the DTFIP graph. Intuitively, nodes in close geographic proximity should tend to be in the same segment of the partition. However, when the DTFIP graph was tested using partitioning algorithms based on geographic proximity of nodes (using "coordinate multi-section" [17]), the solutions were not much better than random solutions. An explanation for this deviation from Simon's findings is that the DTFIP graph contains arcs connecting many nodes that are a great distance apart while several nodes that are close geographically are not connected by arcs. Since coordinate-based initial solutions seem to be no better than randomly produced solutions, a random initial assignment is used in the meta-algorithm designed to solve the DTFIP.

The random assignment procedure scans every node in the graph and assigns it to a partition segment randomly. The resulting partition is then sent to MakeLegal to guarantee that the cardinality constraints are satisfied

and then to Evaluate_Objective which forwards the solution to a specific objective-function evaluator to calculate the objective value of the solution.

The objective-function evaluator is not specifically called by GetInitialSolution or any other algorithm. The algorithm makes a generic call to Evaluate_Objective which in turn forwards the call to a specific objective-function evaluator determined dynamically by the analyst. There are three variants of calls to Evaluate_Objective: The type A variant evaluates the objective value from scratch, while type B and C compute the full objective value efficiently by evaluating the change in the objective value given small changes in the solution. Type B computations are based on changing a single node, and type C computations are based on changing multiple nodes. If Evaluate_Objective is of type B or C and the proposed change is beneficial, the objective-function evaluator makes the change and returns the improved solution and its value. Otherwise the original solution and its value are returned.

In GetInitialSolution and all of the other algorithms, a partition is defined on the nodes as a function v_i where $v_i = k$ if the node i is in the partition k. (In the code, v_i

is represented by an array element v[i].) The pseudo-code for GetInitialSolution is:

```
Procedure: GetInitialSolution (G,M,m,K)
Global variables:   V[], best known solution
                    O, best objective value
Input:              G=(N,A),  an undirected network in
                    adjacency-list form
                    m, minimum partition segment size
                    M, maximum partition segment size
                    K, number of segments
Output:             v[], a random solution
                    o, objective value of the random
                       solution
                    V[], best known solution
                    O, best objective value
{
  For i = 1 to |N| {
    V[i] ← randomInteger(1,K)
                ! randomInteger(a,b) returns a uniformly
                ! distributed random integer from [a,b]
    }
  Call MakeLegal (G,M,m,K,v[])
  o ← Evaluate_ObjectiveA (G,v[])
  If o < O {
    O ← o
    V[] ← v[]
  }
  Return (o, v[])
}
```

Note that Evaluate_ObjectiveA (G,v[]) is a generic function that computes the objective from scratch.

2. Change1Node

This algorithm searches the vicinity of the current solution by sweeping through each node in the graph and moving each node from its current partition segment to each

other segment, in succession. If the solution is feasible, the new partition assignment is sent to an objective-function evaluator for evaluation. The objective-function evaluator makes the proposed change if the solution is improved and returns the modified solution and objective value. If the proposed change does not improve the solution the evaluator returns the original solution and objective value.

The algorithm sweeps through all nodes repeatedly until no better solution is found through an entire sweep of the nodes. The pseudo-code for this procedure is:

```
Procedure:  Change1Node (G,m,M,K,o,v[])
Global variables:   V[], best known solution
                    O, the best known objective value
Input:              G=(N,A),  an undirected network in
                    adjacency-list form
                    m, minimum partition segment size
                    M, maximum partition segment size
                    K, number of segments
                    v[], starting solution
                    o, starting objective value
Output:             v[], solution at algorithm completion
                    o, objective value at algorithm
                       completion

{
  For k = 1 to K {  ! k is the segment of the partition
    c[k] ← 0        ! c[k] will be |N_k|
  }
  Z ← positive infinity
  For i = 1 to |N| {
    k ← v[i]
```

```
    c[k]++
  }
  While Z > o {
    Z ← o
    For i = 1 to |N| {
      k' ← v[i]
      If c[k'] > m {
        For k = 1 to K {
          If k ≠ k' and c[k] < M {
            {o, v[]} ← Evaluate_ObjectiveB (G,o,v[],k,i)
          }
        }
      }
    }
  }
  If o < O {          ! If after algorithm is finished the
                      ! local objective value is better than
                      ! the best objective value, update
                      ! the best solution and objective value
    O ← o
    V[] ← v[]
  }
  Return(o, v[])
}
```

Evaluate_ObjectiveB (G,o,v[],k,i) is a generic function that evaluates the objective of the partition that results from the change in node i's segment from v[i] to k.

3. Swap2Nodes

This algorithm broadens the search from Change1Node, by interchanging two nodes' partition segments simultaneously. The algorithm sweeps through every possible combination of two nodes i ≠ j with v[i] ≠ v[j] and proposes a swap of the partition segments to which they are

assigned. An objective-function evaluator of type C implements the proposed swap and returns the new solution along with the new objective value if it is improved. If no improvement is found, the original solution and objective value are returned to the algorithm. The algorithm keeps proposing node swaps in this fashion until it has looked at every combination without an improvement in the solution.

The concept used by Change1Node and Swap2Nodes may be continued to three or more nodes. These extended algorithms iterate through all possible combinations in the vicinity of the current solution. As the number of nodes interchanged increases, computation time increases exponentially. DTFIP was tested using three-way interchanges through all nodes, but there was little gain in the quality of the solution so it was not used in this thesis. (However, such algorithms will need to be implemented for use with different problems.) The pseudo-code for the two-way interchange procedure is:

```
Procedure:  Swap2Nodes (G,o,v[])
Global variables:   V[], best known solution
                    O, the best known objective value
Input:              G=(N,A),  an undirected network in
                    adjacency-list form
```

```
                    v[], starting solution
                    o, starting objective value
Output:             v[], solution at algorithm completion
                    o, objective value at algorithm
                       completion
                    V[], best known solution
                    O, the best known objective value
{
  Z ← ∞
  While Z > o {
    Z ← o
    For i = 1 to |N| {
      For j = i to |N| {
        If v[i] ≠ v[j] {
          {o, v[]} ← Evaluate_ObjectiveC
                      (G,o,v[],{i,j},{v[j],v[i]})
        }
      }
    }
  }
  If o < O {          ! If after algorithm is finished the
                      ! local objective value is better than
                      ! the best objective value update
                      ! the best solution and objective value
    O ← o
    V[] ← v[]
  }
  Return (o, v[])
}
```

Evaluate_ObjectiveC (G,o,v[],{i,j}, {v[j],v[i]}) is a generic function that evaluates the objective of the partition that results from changing the partition segment of a set of nodes. In this case nodes i and j are moved into segments v[j] and v[i], respectively.

4. RandomizeXNodes

Change1Node and Swap2Nodes search a limited portion of the solution space and quickly find a local optimum. We could use GetInitialSolution to find many different starting locations. Change1Node and Swap2Nodes could then improve on these random solutions until an acceptable solution is found. Although this would work theoretically, it could take a long time. An alternative to getting a new initial solution is to "shock" the best solution.

RandomizeXNodes shocks the solution by randomly changing the partition segment of a pre-specified number of nodes. Once this shock has been performed, there is the possibility that the solution is no longer feasible. To correct this, RandomizeXNodes sends the solution to MakeLegal to check the feasibility of the solution with respect to the cardinality constraints. If the solution is not feasible with respect to the cardinality constraints, MakeLegal makes it so. After feasibility is guaranteed, the solution is sent to an objective-function evaluator of type A. The evaluator determines the new objective value and returns it.

After shocking the system with RandomizeXNodes, Change1Node and Swap2Nodes may be used to improve the solution again.

It is difficult to determine the size and frequency with which these shocks should administered. As was pointed out in Chapter I, the normal procedure is to start with large shocks and reduce the size of the shock slowly. In theory, if the number of nodes changed by these shocks is large enough and that number is reduced slowly enough, the solution will converge [12]. ("Slowly enough" typically implies an exponentially long run time, however.) In the DTFIP meta-algorithms, the initial number of nodes changed by the shocks will be large, say half of the nodes, and will be reduced fairly quickly, say one node per iteration. It is hoped that the solution obtained in this manner will be good, although the optimal solution cannot be guaranteed.

The pseudo-code for this procedure is:

```
Procedure:  RandomizeXNodes (G,m,M,K,o,v[],C)
Global variables:   V[], best known solution
                    O, the best known objective value
Input:              G=(N,A),  an undirected network in
                    adjacency-list form
                    m, minimum partition segment size
                    M, maximum partition segment size
                    K, number of segments
```

```
                    v[], a starting solution
                    o, starting objective value
                    C, number of nodes to change
Output:             v[], a solution with C nodes moved
                    o, objective value after nodes moved
                    V[], best known solution
                    O, the best known objective value
{
  For j = 1 to C {
    i ← randomInteger (1,N)
    v[i] ← randomInteger (1,K)
                ! randomInteger(a,b) returns a uniformly
                ! distributed random integer from [a,b]
  }
  Call MakeLegal (G,m,M,K,v[])
  o ← Evaluate_ObjectiveA (G, v[])
  If o < O {    ! If after algorithm is finished the
                ! local objective value is better than
                ! the best objective value update
                ! the best solution and objective value
    O ← o
    V[] ← v[]
  }
  Return (o, v[])
}
```

(Note: Evaluate_ObjectiveA (G,v[]) is described after the pseudo-code for Change1Node.)

B. OBJECTIVE-FUNCTION EVALUATORS

Objective-function evaluators receive a proposed solution from an algorithm via Evaluate_Objective and compute the solution's objective value. If the objective value is improved, the current solution is updated. Objective-function evaluators must be able to work with a variety of solution changes. If they always recalculate

the objective value from scratch, much computational effort will be wasted. To keep the solver running as fast as possible, three versions of each objective-function evaluator are used: Type A objective-function evaluators evaluate the objective value from scratch, while type B and C compute the full objective value efficiently by evaluating the change in the objective value given small changes in the solution. Type B computations are based on changing a single node, and type C computations are based on changing multiple nodes.

1. ObjectiveCountInterfaceNodes

The objective-function evaluator ObjectiveCountInterfaceNodes evaluates the function for Problem 1. It counts the total number of interface nodes that will be needed in the graph. As is required by all objective-function evaluators, it will evaluate changes to just one node, a set of nodes, or re-evaluate the entire solution. The objective-computing procedures are:

Procedure: ObjectiveCountInterfaceNodesA (G,v[])
Scans every node i in G. If i is adjacent to one or more nodes not in the same segment as i, then i is an interface node.
Input: G=(N,A), an undirected network in adjacency-list form
v[], solution to be evaluated

```
Output:              o, current objective value which is the
                     number of interface nodes in the
                     solution defined by v[]
{
  o ← 0
  For i = 1 to |N| {
    found ← false
    For each node j adjacent to i {
      If v[i] ≠ v[j] {
        If found = false {
           o++
        }
        found ← true
      }
    }
  }
  Return o
}
```

Procedure: ObjectiveCountInterfaceNodesB (G,o,v[],p,i)
Checks if input node i is an interface node in its current partition segment and the proposed segment N_p. If it is an interface node in its current segment but is not if moved to segment p, the node is moved and the current objective value is reduced by one; otherwise the original partition and its objective value are returned.

```
Input:               G=(N,A),  an undirected network in
                     adjacency-list form
                     v[], current solution
                     o, current solution value which is the
                        number of interface nodes in the
                        partition defined by v[]
                     i, a node proposed to be moved to Np
                     p, a proposed partition segment index
                        for i
Output:              v[], the input solution if the proposed
                          move does not improve the
                          objective value; otherwise the
                          input solution with node i moved
                          to segment p
                     o, objective value of returned solution
{
```

```
  found ← false
  better ← true
  For each node j adjacent to i {
    If v[i] ≠ v[j]{
      found ← true
    }
    If p ≠ v[j]{
      Better ← false
    }
  }
  If found and better {        ! if the objective value is
                               ! improved by the move
    o--
    v[n] ← p
  }
  Return (o,v[])
}
```

Procedure: ObjectiveCountInterfaceNodesC
(G,o,v[],set S,mapping p[])

Counts the number of interface nodes, if any, that would be reduced by moving each node i ∈ S from its current partition segment to a new partition segment $N_{p[i]}$. If the number of interface nodes is reduced by the proposed moves, the solution is modified by making these moves and the objective value is updated; otherwise the original partition and its objective value are returned.

Input: G=(N,A), an undirected network in adjacency-list form
v[], current solution
o, current solution value which is the number of interface nodes in the partition defined by v[]
S, a set of nodes
p[], a mapping of nodes to partition segments

Output: v[], the input solution if the proposed moves do not improve the objective value; otherwise, the input solution with each node s ∈ S moved to p[s].

```
                    o, objective value of returned solution
{
  count ← 0
  found ← false
  better ← true
  for each i ∈ S {
    found ← false
    For each node j adjacent to i {
      If v[i] ≠ v[j] {
        found ← true
      }
      If p[i] ≠ v[j]{
        better ← false
      }
    }
    If found and better {
      Count++
    }
    If not found and not better {
      Count--
    }
  }
  If count > 0 {      ! count is the amount by which the
                      ! proposed change will improve the
                      ! objective value
    For each s in S {v[s] ← p[s]}
    o ← o - count
  }
  Return (o,v[])
}
```

2. ObjectiveInterfaceNodesInOrder

The objective-function evaluator ObjectiveInterfaceNodesInOrder evaluates the objective function for Problem 2: That problem is to minimize the peak number of interface hardware sets that will be needed over all steps of DISA's upgrade. This objective-function

evaluator has three subroutines analogous to those for ObjectiveCountInterfaceNodes: It will evaluate changes to just one node, a set of nodes, or re-evaluate the entire solution.

For Problem 2, even a change in a single node can affect the status of several nodes. Because the potential number of nodes affected by the change is large, an efficient means of reducing the computational effort has not been determined. Thus, all objective function evaluations are essentially made from scratch for Problem 2. If a more efficient means is determined, it can be implemented by modifying the type B and C variants of ObjectiveCountInterfaceNodes. The pseudo-code for these procedures are:

```
Procedure:  ObjectiveInterfaceNodesInOrderA (G v[])
     For each partition segment k < K (for each step of the
upgrade process), every node i in G is scanned.  If node i
is adjacent to one or more nodes j such that v[j] > k, and
v[i] ≤ k, i is an interface node for that step of the
upgrade.  The objective value is the largest number of
interface nodes found at any step.
Input:              G=(N,A),  an undirected network in
                    adjacency-list form
                    v[], solution to be evaluated
Output:             o, objective value of returned solution
{
  o ← 0
  current ← 0
  worst ← 0
```

```
  K ← max_{i=1,…,|N|}  v[i]
  For k = 1 to K-1 {
    current ← 0
    For i = 1 to |N| {
      If v[i] ≤ k {
        found ← false
        For each node j adjacent to i {
          If v[j] > k {
            found ← true
          }
        If found {current++}
        }
      }
    }
    If current > worst {worst ← current}
  }
  o ← worst
  Return o
}
```

Procedure: ObjectiveInterfaceNodesInOrderB (G,o,v[],p,i)
Moves node i to proposed segment p and computes the Problem 2 objective as in ObjectiveInterfaceNodesInOrderA. If the objective value for the proposed change is an improvement, the move is made permanent and the solution and its objective value are updated; otherwise the original partition and its objective value are returned.

Input: G=(N,A), an undirected network in adjacency-list form
v[], current solution
o, current solution value which is the number of interface nodes in the partition defined by v[]
i, a node proposed to be moved to N_p
p, proposed partition segment for i

Output: v[], the input solution if the proposed move does not improve the objective value, the input solution with node k moved if the move does improve the solution
o, objective value of returned solution

{

```
  c ← v[i]
  v[i] ← p
  current ← 0
  worst ← 0
  K ← max_{i=1,…,|N|}  v[i]
  For k = 1 to K-1 {
    current ← 0
    For i' = 1 to |N| {
      If v[i'] ≤ k {
        found ← false
        For each node j adjacent to i' {
          If v[j] > k {
            found ← true
          }
        If found {current++}
        }
      }
    }
    If current > worst {worst ← current}
  }
  If worst < o {o ← worst}
  else {v[i] ← c}
  Return {o, v[]}
}
```

Procedure: ObjectiveInterfaceNodesInOrderC
(G,o,v[],set S,mapping p)

Moves each node i ∈ S from its current partition segment to a new partition segment $N_{p[i]}$. Then computes the Problem 2 objective value as in ObjectiveInterfaceNodesInOrderA. If the objective value for the proposed change is an improvement, the move is made permanent and the solution and its objective value are updated; otherwise, the original partition and its objective value are returned.

Input: G=(N,A), an undirected network in adjacency-list form
v[], current solution
o, current solution value which is the number of interface nodes in the partition defined by v[i]
S, a set of nodes

```
                    p, a mapping of nodes to partition
                         segments p[i]
Output:             v[], the input solution if the proposed
                         move does not improve the
                         objective value; otherwise, the
                         input solution with nodes in S
                         moved
                    o, objective value of returned solution
{
  for each s in S {c[s] ← v[s]; v[s] ← p[s]}
  current ← 0
  worst ← 0
  K ← max_{i=1,…,|N|} v[i]
  For k = 1 to K-1 {
    current ← 0
    For i = 1 to |N| {
      If v[i] ≤ k {
        found ← false
        For each node j adjacent to i {
          If v[j] > k {
            found ← true
          }
        If found {current++}
        }
      }
    }
    If current > worst {worst ← current}
  }
  If worst < o {o ← worst}
  else {for each s in S; v[s] ← c[s]}
  Return {o, v[]}
}
```

C. MAKELEGAL

Any algorithm that has a possibility of violating the cardinality constraints $m \leq |N_k| \leq M$ for $k = 1,\ldots, K$ calls MakeLegal. If $|N_k| < m$ for any segment k, nodes are repeatedly moved from the largest segment to k. If $|N_k| > M$

for any segment k, nodes are repeatedly moved to the smallest segment from k. For this thesis two algorithms call MakeLegal, namely GetInitialSolution and RandomizeXNodes. The pseudo-code for MakeLegal is:

```
Procedure:     MakeLegal (G,m,M,K,v[])
     Verifies that cardinality constraints are met.  But,
if |N_k| < m for any k, a node is moved from the largest
segment to that segment.  And, if |N_k| > M for any k, a node
is moved to the smallest segment from k.
Input:         G=(N,A),  an undirected network in
               adjacency-list form
               m, minimum partition segment size
               M, maximum partition segment size
               K, number of partition segments
               v[], current solution possibly infeasible
Output:        v[], feasible solution

{
  c[k] ← 0 for k = 1,…,K      ! c[k] will be |N_k|
  For i = 1 to |N| {
    k ← v[i]
    c[k]++
  }
  z ← argmax_{k=1,…,K} c[k]    ! N_z is the largest
                               ! partition segment

  For k = 1 to K {
    While c[k] < m {
      j ← random node with v[j] = z
      v[j] ← k
      c[k]++
      c[z]--
      z ← argmax_{k=1,…,K} c[k]
    }
  }
  y ← argmin_{k=1,…,K} c[k]    ! N_y is the smallest
                               ! partition segment
  For k = 1 to K {
```

```
    While c[k] > M {
      j ← random node with v[j] = k
      v[j] ← y
      c[y]++
      c[k]--
      y ← argmin_{k=1,…,K} c[k]
    }
  }
  Return v[]
}
```

IV. A DEMONSTRATION OF DORS

This chapter demonstrates the operation of DORS. The purpose is to illustrate the versatility of DORS through the use of the library of algorithms and objective-function evaluators introduced in Chapter III. This library will be applied to the DTFIP and provide solutions using a 266 MHz Pentium II PC operating under Windows NT 4.0.

To avoid confusion, the British standard of placing punctuation outside of quotes is adopted in this chapter. The items inside of quotes represent the exact form of a word, phrase, or file name used with DORS.

A. GRAPHICAL USER INTERFACE

When DORS is started, a Graphical User Interface (GUI) is initiated to provide all the major functions of DORS with simple mouse and keyboard commands. The initial screen of the GUI is shown in Figure 1 and a brief description of all of the functions of DORS follows here.

1. Graph Control

The graph control section of the startup window consists of the four buttons in the upper left hand corner of Figure 1 labeled "Change Arc File", "Change Node File", "Save Arc File", and "Save Node File", along with the four

lines of text immediately to the right of those four buttons.

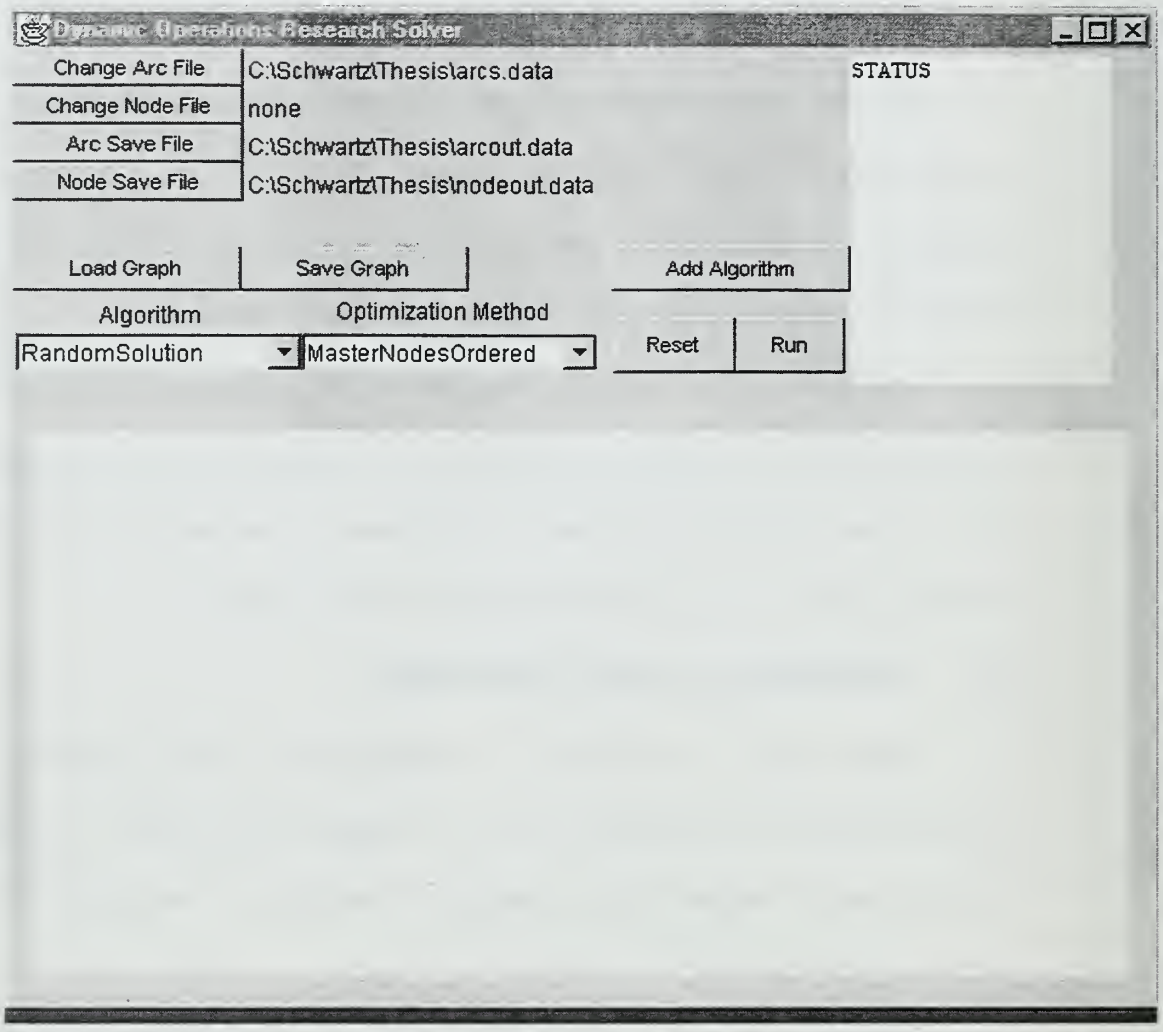

Figure 1. Initial screen of DORS

The graph control section determines which files will be used for loading and saving graphs. The files are space-delimited, a standard form of output that is recognized by all major spreadsheets and word processors.

This makes data manipulation trivial once the data is in the proper format; the graphs may be loaded into a spreadsheet for easy sorting and displaying of the results of DORS.

DORS can load a graph using either an arc set, a node set, or both. These files are chosen in the graph control section. The files must follow a standard format. This format includes a header line that describes the elements contained in the file body followed by the body of the file that may contain any number of lines to represent either the arcs or nodes of the graph.

a. Change Node File Button

When the mouse is clicked over the Change Node File button a second window is accessed as shown in Figure 2. This window allows the user to browse through local data storage media such as diskettes, hard drives, or CD-ROMs to find a file representing a node set. The user may also type in the filename and location of a file to access the data on a local storage device.

The filename of the current file containing the node set is displayed immediately to the right of the Change Node File button. If this file name is changed to

"none" or made null by entering nothing, the solver will not load a node set.

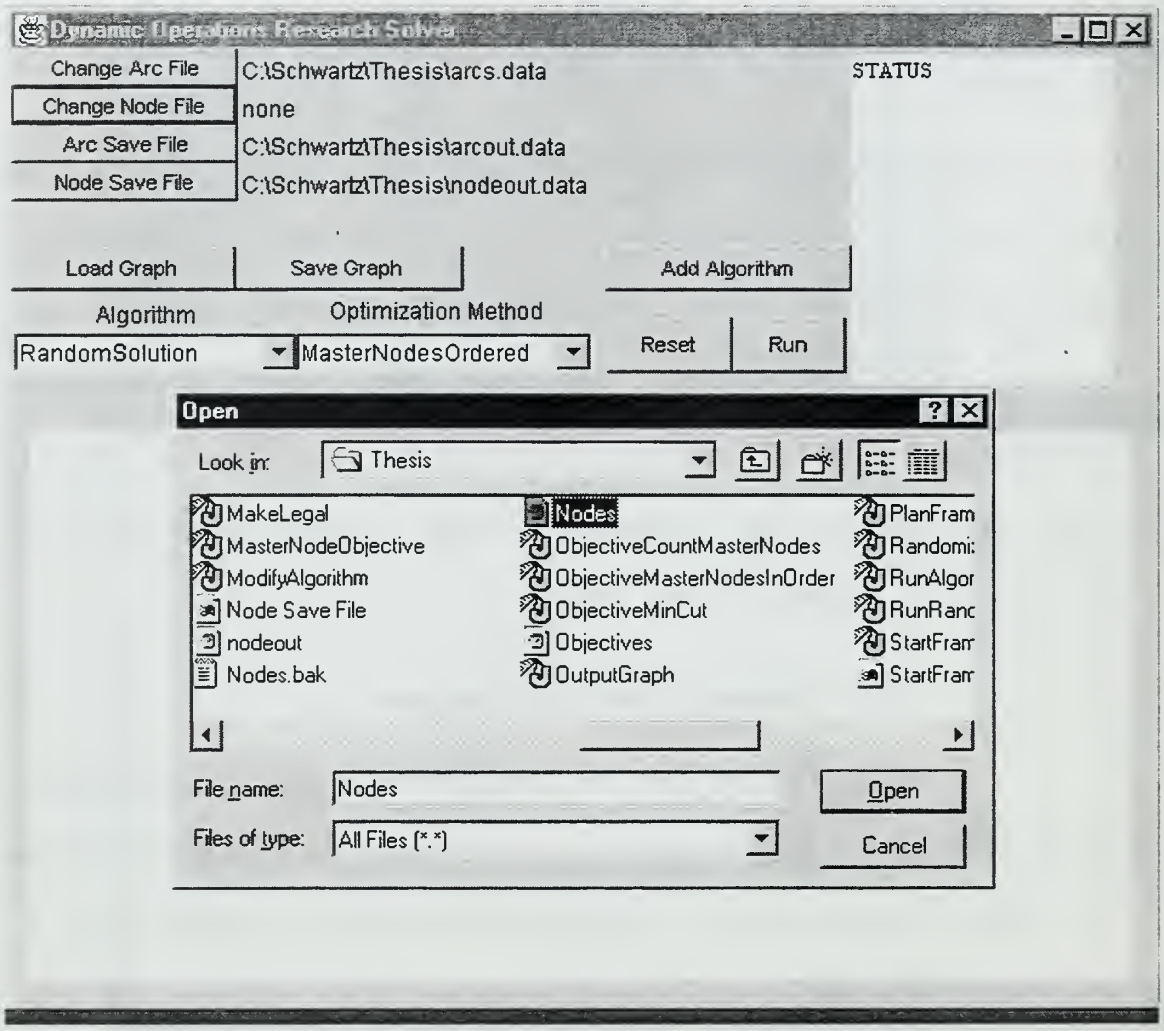

Figure 2. File browsing window in DORS

The first few lines of the file with the node set used for DTFIP follow:

name java.lang.String degreesLatitude java.lang.Double degreesLongitude java.lang.Double

D001N184	24.5517	-81.7967
D001N137	24.5667	-81.6833
D001N175	25.9167	-80.2667
D001N110	27.6906	-97.2894

In the example above, the header line wraps over two lines. The first column contains a property of the node called "name" which is a String data type. A String data type in Java is simply text. The name is used for unique identification of the node. The second and third columns contain properties labeled "degreesLatitude" and "degreesLongitude" of data type Double (a floating-point, double-precision number). These two properties define the physical location of the node in degrees North Latitude, and degrees East Longitude.

b. Change Arc File Button

The Change Arc File button works identically to the Change Node File button. A sample from the first few lines of a file containing the DTFIP arcs follows:

name java.lang.String name java.lang.String flow java.lang.Integer

D001N001	D001N013	1
D001N001	D001N028	1

D001N001	D001N037	1
D001N001	D001N043	1
D001N001	D001N052	1
D001N001	D001N123	2
D001N003	D001N004	1

The first column is a property called "name" associated with the head node of the arc and the second is the "name" property for the tail arc. The naming convention here is critical. The property used to identify nodes in the node file and both head and tail nodes in the arc file must be the same. If the properties do not have the same name, DORS will think each of these is a separate node and the graph will not be loaded correctly. In DTFIP the property name is used to identify the node.

The final column is a property called "flow", data type Integer. This number represents the number of transmission lines in the arc and is not used in the DORS algorithms.

c. Save Node File and Save Arc File

The Save Node File and Save Arc File buttons allow the user to change the files to which the graph is saved. Functionally, they work exactly like the Load Node File and Load Arc File buttons and create files that are

very similar. These files hold the graph after it has been modified by DORS and properties have been added.

2. Load Graph and Save Graph Buttons

The Load Graph and Save Graph buttons are located directly below the Graph Control Section of the window shown in Figure 1. The Load Graph button loads a graph into the solver as a Konig graph [16] by accessing the files listed to the right of the Change Node File and Change Arc File buttons. This action must be taken prior to running a meta-algorithm. Otherwise, the meta-algorithm can perform no actions.

The Save Graph button accesses the graph in Konig and saves an arc set and a node set to the files indicated to the right of the Arc Save File and Node Save File buttons. It is designed to save a copy of the graph and a problem solution after DORS has found a (candidate) solution. It saves the name and all properties associated with the graph and the solution.

3. Algorithm Selector

The Algorithm Selector is located directly under the Load File button and is accessed by clicking the mouse over the arrow to the right of the white field. The white field

contains the name of the algorithm that is currently selected. When the user clicks the arrow on the right, a list is displayed as shown in Figure 3.

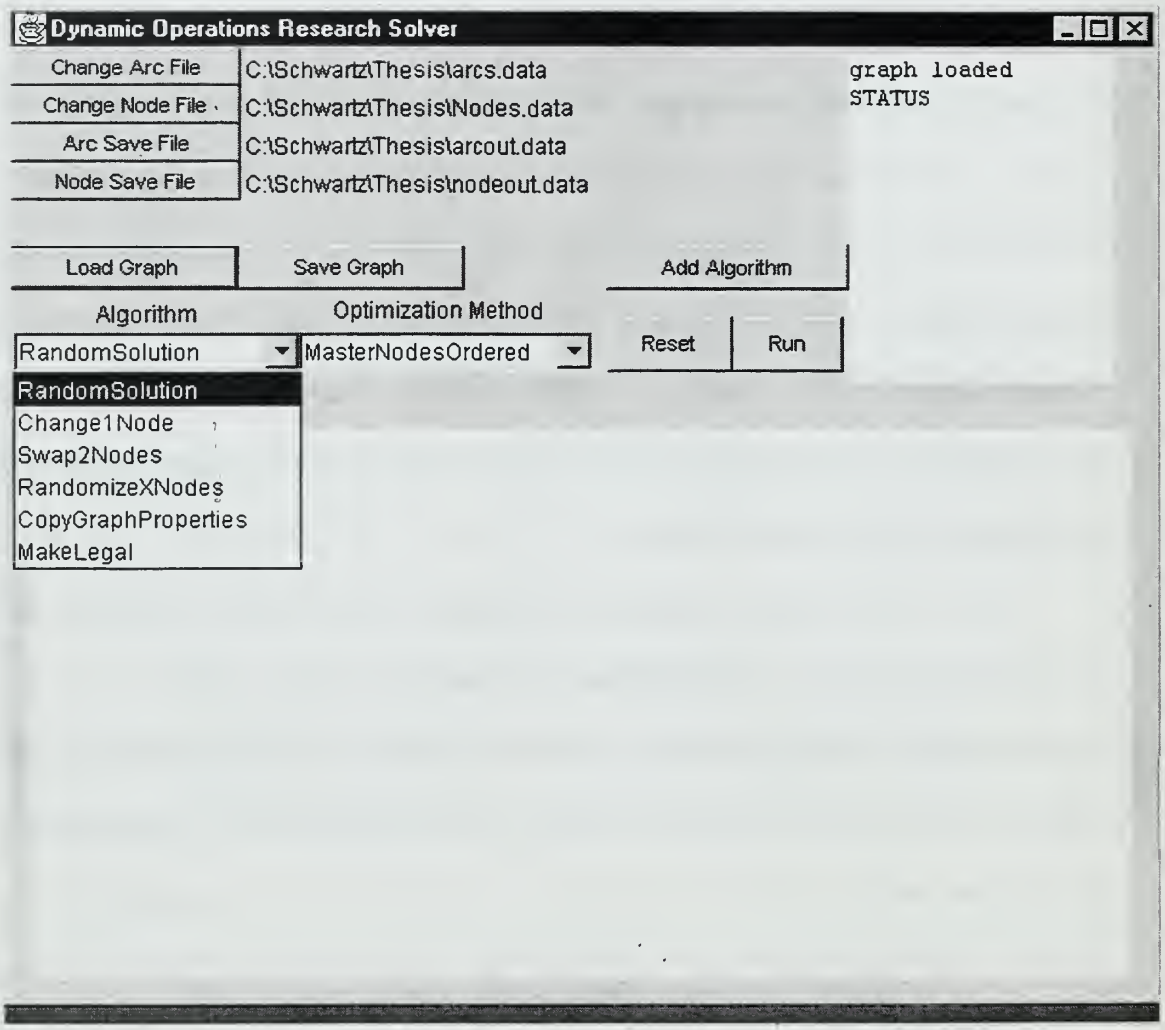

Figure 3. Selecting an algorithm in DORS

Algorithms may be added to DORS by placing the algorithm name and location in the list of algorithms contained in the file "algorithms.data".

4. Reset Button

The Reset Button is located on the left just underneath the Add Algorithm button. It simply empties the meta-algorithm leaving no algorithms to run.

5. Optimization Method Selector

The Optimization Method Selector operates in the same manner as the Algorithm Selector: Click the mouse on the arrow to the right of the selector and highlight the name of the desired objective-function evaluator with the mouse. The selected objective-function evaluator will be used by the meta-algorithm.

Objective-function evaluators maybe added by placing the name and location of the evaluator in the file objectives.data.

6. Add Algorithm Button

The Add Algorithm button is to the right and slightly above the Optimization Method Selector. It queries the algorithm selected in the Algorithm Selector to find out what inputs the algorithm needs. DORS then opens a query window and asks the user to identify the inputs to the algorithm. The window contains default values that the user can change (see figure 4).

After the Save button is pressed at the bottom of the query window, the window is closed and the algorithm is added to the meta-algorithm.

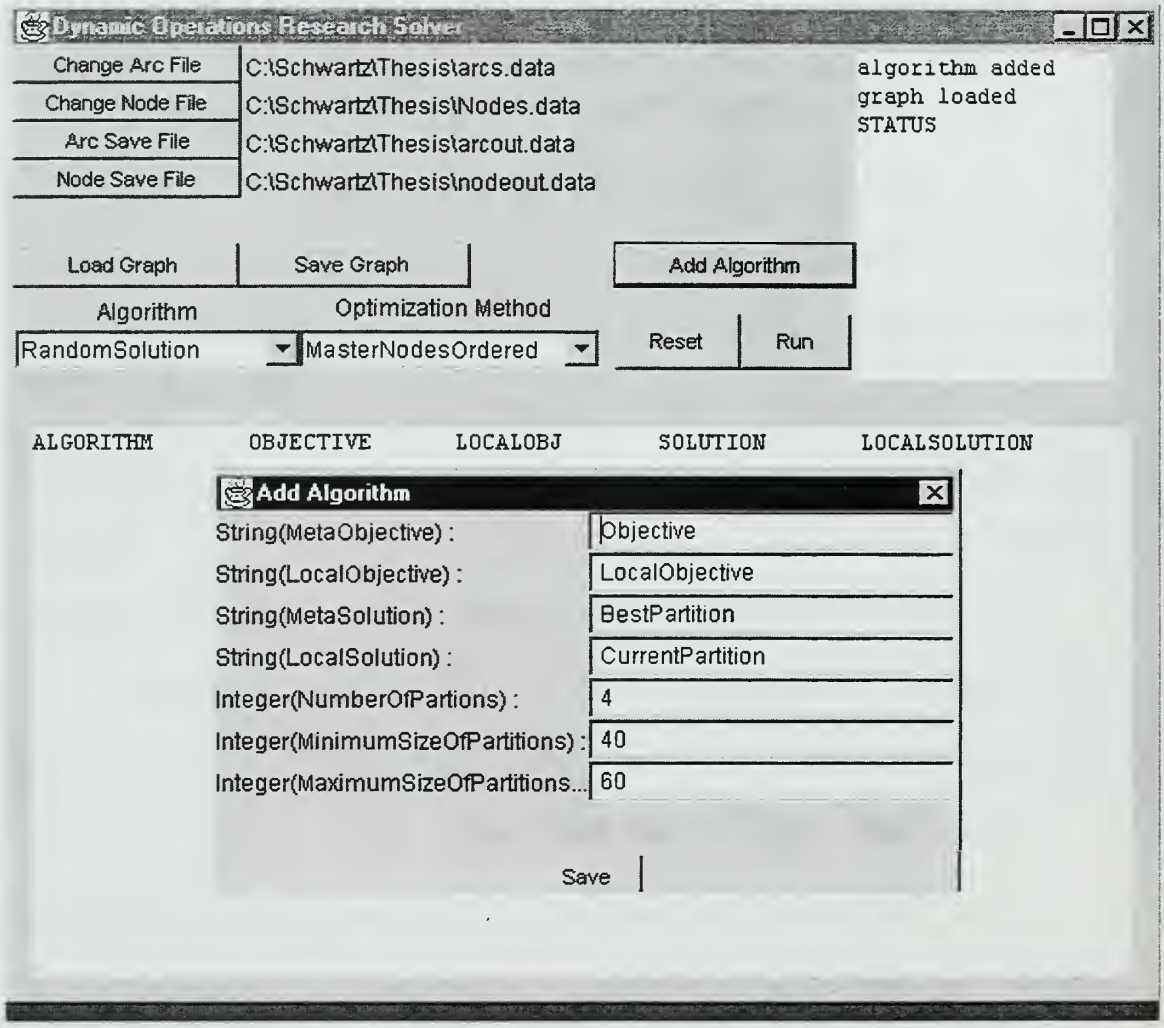

Figure 4. Adding an algorithm to the meta-algorithm

Each time the Add Algorithm button is used, a new query window appears and the meta-algorithm has the algorithm displayed in the Algorithm Selector added to it. The meta-

algorithm will step through each algorithm in the order added until each algorithm has been run. The solver will not loop through algorithms; a loop is simulated by repeatedly listing the algorithms from within the desired loop.

7. Run Button

Placing the cursor over the Run button and clicking the mouse button causes DORS to run the current meta-algorithm. The objective-function evaluator displayed in the Optimization Method selector is used for all the algorithms in the meta-algorithm. The same meta-algorithm may be run more than once by waiting until it is done and then clicking again. The analyst can dynamically change the meta-algorithm through the use of the Run Algorithm button. The objective function may be changed by selecting a new objective-function evaluator in the Optimization Selector and clicking the Run button again.

8. Status Box

The Status Box is the smaller white field in the upper right of the window; the word "STATUS" is displayed when the program is started. This area informs the user when key DORS functions are completed so that other functions

may be used. The information this area gives the user is: When graphs are loaded or saved, when algorithms are added, and when each step of the meta-algorithm is complete. Without this box, it would be difficult for the user to tell when long-running functions are completed.

9. Meta-Algorithm Box

The Meta-Algorithm Box is the white field in the bottom of the window. It contains key data about each of the algorithms in the current meta-algorithm. These five key fields give the name of the algorithm, the name assigned to best solution, the name of the solution being manipulated by the algorithm, the name assigned to the value of the best solution, and the name assigned to the solution being manipulated by the current algorithm. These five pieces of data allow the user to quickly ascertain how solutions will be constructed as the meta-algorithm steps through the list of algorithms. An example of the data displayed in the Meta-Algorithm box is shown in Figure 5.

10. Graph Panel

When a run of the meta-algorithm is complete, the Graph Panel is displayed. The graph panel allows the user to look at the properties of any node, arc, or of the graph

itself. The top box of Figure 6 displays the properties associated with the graph. It contains the various values, such as the objective value, that are associated with the graph. The lower boxes contain the properties of the nodes and arcs of the graph. Any node or arc may be selected in the same manner that algorithms and objective-function evaluators are selected in the main window. In this case, however, the information is displayed in the box directly below the selector associated with it. The key information contained in these boxes is the current solution. The segment of any partition may be easily ascertained here.

If the data needs to be looked at more closely and a larger quantity of data needs to be displayed at the same time, the same information can be obtained in a compact form by clicking the Save Graph button and then viewing the file with a text editor or spreadsheet.

The Graph Panel provides a means to check key values of the solutions after a meta-algorithm run is complete. Graph Panel is constructed by Konig and is completely external to DORS.

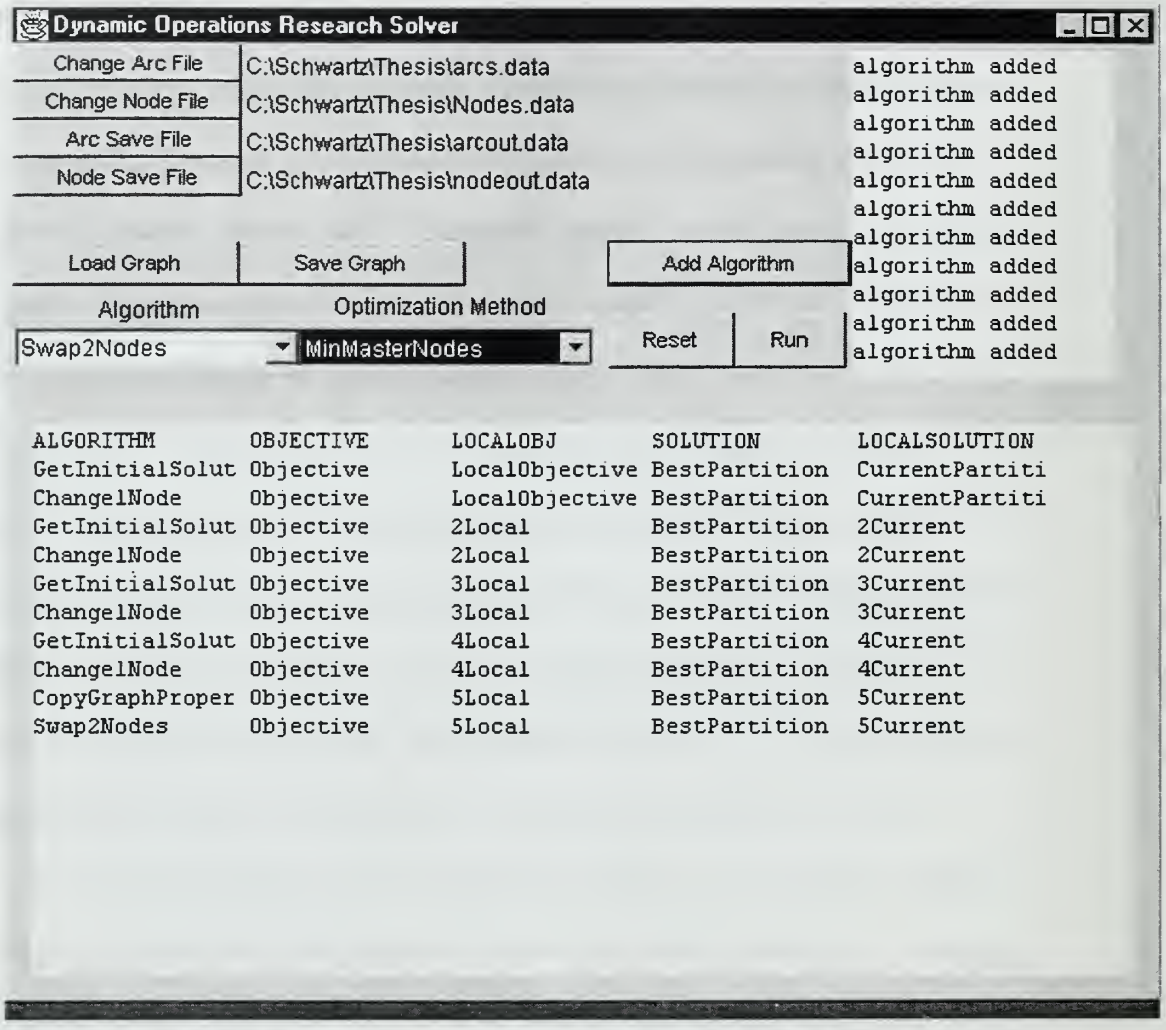

Figure 5. Sample Meta-Algorithm Box output for DORS

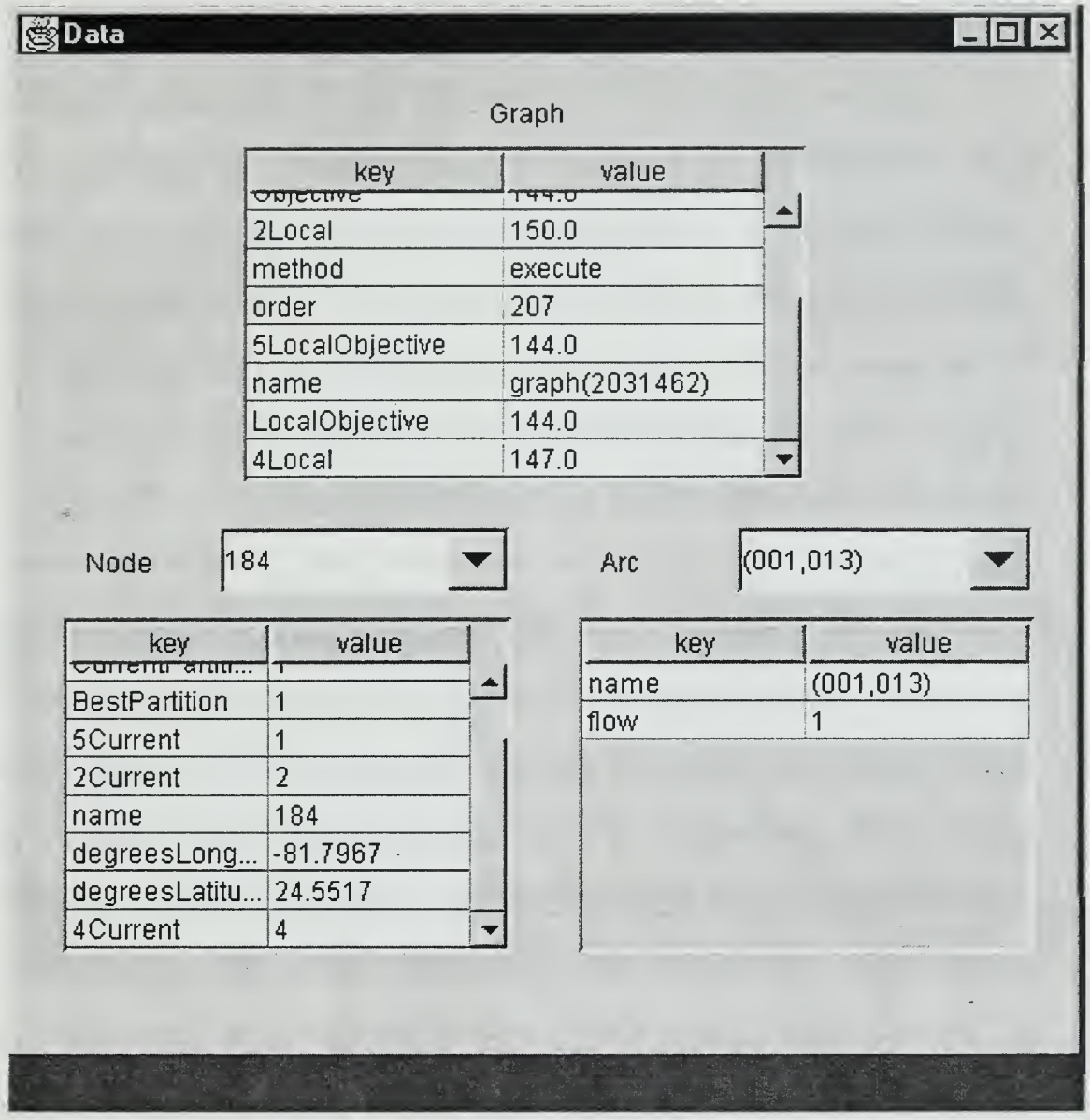

Figure 6. Graph Panel display provided by Konig

B. EXAMPLE OF A META-ALGORITHM RUN WITH DORS

This is a simple walk through of a DORS meta-algorithm run, from start to finish. First, start DORS by running the program file "DynamicSolver.class". This may be done through the normal method of launching a Java application. In Windows 95,98, or NT the application is launched by typing "java Thesis.DynamicSolver" from the DOS prompt. This will load DORS and the window in Figure 1 will be displayed.

Now that DORS is running, load a graph. We will load the graph used in the DTFIP. Use the mouse to place the cursor over the button labeled "Change Arc File" and click the left mouse button. A window labeled "Open" appears on the monitor, as in Figure 2. This window provides two methods for selecting the file that contains the list of arcs. You may select a file by placing the cursor over the file name and clicking the left mouse button twice or you may type the file name in the box labeled "File name" followed by placing the cursor over the button labeled "Open" and clicking the left mouse button. Use the second method and type "arcs.data" in the box labeled "File name".

Then place the cursor over the Open button and click the left mouse. Now the file is selected.

Load the selected file as a graph by placing the cursor over the Load Graph button and clicking the left mouse button. When the graph is loaded, the words "graph loaded" appear in the Status Box.

Now we will create a meta-algorithm. The first algorithm is GetInitialSolution. Since the Algorithm Selector already has this showing, place the cursor over the Add Algorithm button and click the left mouse button. A window similar to the one in Figure 4 appears. The parameters of the algorithm being added may be changed by clicking on the field that you wish to change and typing the desired data. For this demonstration do not change this data. Now accept the data in the window; place the cursor over the button labeled "Save" and click the left mouse button. The first algorithm is now loaded and appears in the Meta-Algorithm box.

For adding more algorithms, we follow the same procedure. Now we will add ChangeINode. Since the Algorithm Selector has GetInitialSolution showing we need to change it. Place the cursor over the arrow just to the left of the word GetInitialSolution in the Algorithm

Selector. The Algorithm Selector will display the algorithms available as in Figure 3. Place the cursor over Change1Node and click the left mouse button. The list disappears and Change1Node is displayed in the Algorithm Selector. Now place the cursor over the Add Algorithm button and click the left mouse button to bring up the Algorithm Properties window. Once again the properties are correct, so click on the Save button to add the second algorithm to the meta-algorithm. Change1Node now appears in the Meta-Algorithm box.

Look at the Meta-Algorithm box closely now. Notice that the four columns labeled "OBJECTIVE", "LOCALOBJ", "SOLUTION", and "LOCALSOLUTION" each have identical entries in them. These are the values entered in the Algorithm Properties window and correspond to the names of the best objective value, the local objective value, the best solution, and the local solution. We will use this information in the next algorithm.

Now we want to add RandomizeXNodes to shock the solution. Use the Algorithm Selector to chose RandomizeXNodes and click the Add Algorithm button to display the Algorithm Properties window. This time we wish to change the properties so the algorithm operates on the

best solution and objective value. Click on the box labeled "String(MetaAlgorithm)" and change the text to "Objective2". Pay attention to the case because Java is case sensitive. Change the text in the next box labeled "String(LocalObjective)" to "Objective". Change the text in the third box labeled "String(MetaSolution)" to "Best2". Change the text in the fourth box labeled "String(LocalSolution)" to "BestPartition".

These changes select the best solution and objective value to be used by RandomizeXNodes. This change in the Algorithm Properties window made the best solution and best objective value into the local solution and local objective value. Copies of the best solution and best objective values were made under the property names just assigned. Now click on the save button and note that the third algorithm displayed in the Meta-Algorithm box has the property names we just assigned listed next to RandomizeXNodes in the third row of data.

We now run the meta-algorithm. Place the cursor over the Run button and click the left mouse button. The meta-algorithm is now running. When the meta-algorithm is complete, a window with the Graph Panel is displayed.

The top section labeled "Graph" now has the graph properties displayed. These include the name of the graph assigned by DORS, the order of the graph(number of nodes), the size of the graph (number of arcs), and the properties we have assigned in DORS. These properties are "LocalObjective", "Objective", and "Objective2", each of which has the objective value associated with the solutions displayed in the node section of the Graph Panel.

The bottom left of the Graph Panel displays the properties associated with each node. A node may be selected by placing the cursor over the box next to the word "Node" and clicking the left mouse button to bring up the list of nodes. A node is selected by a second click of the left mouse button on the node name. The list of properties for the node selected is displayed. The properties for each node are "name" and the properties we assigned during the meta-algorithm creation: "CurrentPartition", "BestPartition", and "Best2".

The last section of the Graph Panel displays the properties associated with the arcs. The display is similar to the node section.

Now we save the solutions to files. Select the name of the file to save the list of arcs and the associated

properties by placing the mouse over the Arc Save File and clicking the left mouse button. A window appears as in Figure 2. Type "arc2.data" in the box labeled "File name" and click on the Open button. "arc2.data" now appears to the right of the Arc Save File button. Follow the same procedure for selecting the file to save the nodes and the associated properties in the same manner with the Node Save File button. This time use the file name "nodes2.data". Now that the file names have been selected, place the cursor over the Save Graph button and click the left mouse button. The data has been saved to the files and you have successfully completed a session with DORS.

V. ANALYSIS OF RESULTS

DORS is used here to solve, heuristically, both formulations of DTFIP presented in Chapter III. DISA would like the number of partition segments K to be four or five and they would like $\delta \approx 10$. An alternative δ of fifteen is considered to determine how relaxation of the partition size constraints affects solutions.

The same meta-algorithm is used for multiple runs using both objective functions described in Chapter II, and variations of K and δ. The meta-algorithm starts with five random solutions provided by GetInitialSolution, each followed by Change1Node for refinement. This provides five separate solutions: The best solution is kept and the other four are discarded by the meta-algorithm. Next, Swap2Nodes is used on the best solution found so far. After Swap2Nodes, 51 repetitions of RandomizeXNodes, each followed by Change1Node, are run. This is done to shock the system as in Chapter III. The first shock changes 100 nodes randomly and each subsequent shock reduces the number by one until the last shock changes only 50 nodes. After the final shock is complete and Change1Node has found a

local optimum, the best solution found to this point is sent to Swap2nodes and the meta-algorithm concludes.

The results are presented in the format shown in Table 1. The complete meta-algorithm contains 114 invocations of the four algorithms. Because of the number of invocations in the meta-algorithm, only selected steps are shown.

Algorithm Number	Selected step in algorithm	Adjusted Cumulative Seconds	Objective Value
1	GetInitialSolution (1)		
2	Change1Node (1)		
3	GetInitialSolution (2)		
5	Change1Node (2)		
7	Change1Node (3)		
9	Change1Node (4)		
11	Change1Node (5)		
12	Swap2Nodes (1)		
13	RandomizeXNodes (First Instance)		
33	RandomizeXNodes (Eleventh Instance)		
53	RandomizeXNodes (Twenty-first Instance)		
73	RandomizeXNodes (Thirty-first Instance)		
93	RandomizeXNodes (Forty-first Instance)		
114	Swap2Nodes (2)		

Table 1. Example of objective function values and adjusted cumulative time for selected steps of the meta-algorithm.

The times recorded in the tables are "adjusted." Because DORS was built using JBuilder© [19] and is not yet 100 percent platform-independent. It must be run with the JBuilder run-time system which is very slow. When the GUI is removed and a state-of-the-art run-time system is used,

the execution time decreases by a factor of about 117 over JBuilder's times. To reflect the speed that should be available when the GUI interface is rebuilt without JBuilder, the actual execution times are divided by 117.

A. First Formulation

Problem 1 (as well as Problem 2) has four variants. K is set at both four and five and δ is changed from ten to fifteen by varying m and M. Four solutions are provided to give DISA multiple options should they decide to implement one of them. For Problem 1, all four variants provide poor solutions.

Problem 1 attempts to minimize the number of interface nodes. The solution for each variant of the problem has an objective value of at least 188; this leaves fewer than 20 nodes that are not interface nodes in the solution. The solutions provided under this objective are not much better than simply designating every node an interface node.

In addition to being poor, the solutions have little commonality among them. They appear to be randomly generated solutions. This seemingly random generation of solutions combined with the lack of real improvement in the objective value bring into question the validity of the

meta-algorithm for this objective function: As designed, perhaps the meta-algorithm simply cannot find a good solution, but better solutions do exist.

However, a second explanation for poor solutions may lie in the structure of the DII. It could be that there are no good solutions for Problem 1, and, in fact, the solutions provided are close to optimal. This idea is supported by noting the average degree of the nodes, at fourteen, is large and by noting that the arcs tend to connect distant nodes as often as close nodes.

To help decide which explanation is correct, the meta-algorithm needs to be tested. For the first test, the graph in Figure 7 with 20 nodes and 36 arcs is created. This graph is small enough that an optimal solution can be found by hand.

The problem parameters for the meta-algorithm are modified to conform to the size of the test graph by setting $K = 4$, $m = 3$, $M = 8$. The meta-algorithm, with these modified parameters, finds the optimal objective value of seven in each of ten test runs performed on this graph. This indicates that the meta-algorithm may be doing its job.

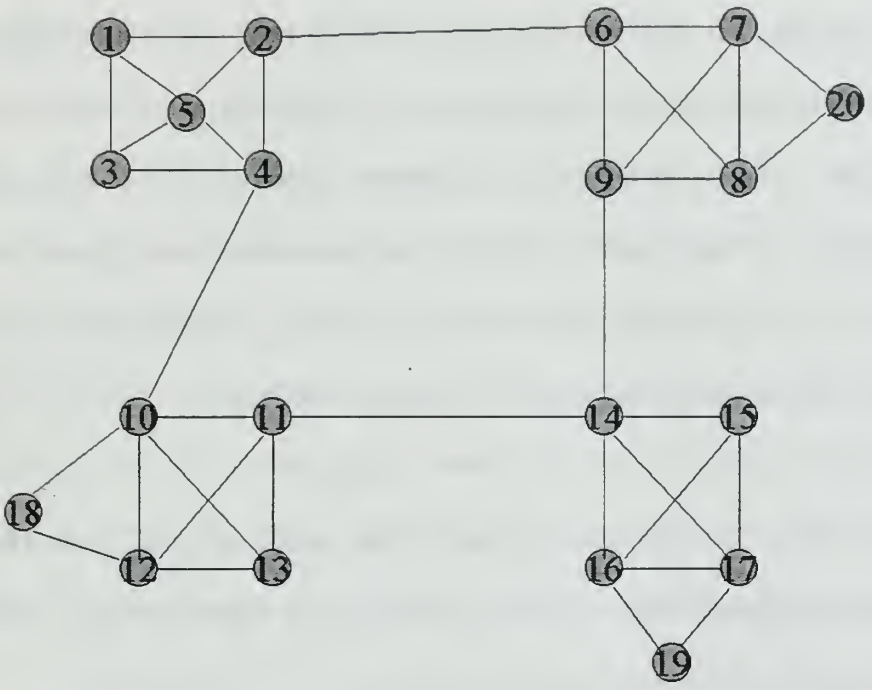

Figure 7. Test graph with 20 nodes and 36 arcs

But the meta-algorithm needs to be tested on a larger problem. Four subgraphs with 50 nodes and 350 arcs each are randomly created for this test. The subgraphs are formed by making each node in the subgraph the end point of seven arcs, and then randomly selecting seven other nodes in the subgraph for the other end point of each of those arcs. The four subgraphs are then connected by four arcs so that each subgraph is connected to two other subgraphs

with a single arc and the resulting graph is connected. In this large graph an objective value of seven is reached by placing each subgraph in a segment. This may not be the optimal objective value but is highly likely to be because of the graph's structure. If the meta-algorithm finds an objective value of seven or lower it has, probably, found the optimal objective value and has performed satisfactorily.

To test the large test graph, the meta-algorithm is given parameters typical of the DTFIP, in particular, K = 4, m = 40, and M = 60.

The meta-algorithm is much less successful on the larger graph than on the smaller one. The best objective value found by the meta-algorithm is 132 which is far from the upper bound of seven. When looking at the meta-algorithm's solution it is noted that the graph is segmented into small groups of nodes. It is not possible to move one or two nodes from these small groups without increasing the objective value. The meta-algorithm breaks these groups through the use of RandomizeXNodes, but the nodes re-form into new groups that remain unbreakable in Change1Node and Swap2Nodes.

A different algorithm is needed to work with these node groups. An algorithm that moves large blocks of nodes together may be the proper tool. (A genetic algorithm might fit this bill. For example, see [18].) It is possible that such an algorithm would combine the small isolated groups of nodes being left by the meta-algorithm and provide improved solutions. Once a few groups are combined, a local search algorithm such as Change1Node may be effective again. Since such an algorithm does not currently exist in DORS, the small groups are not properly dealt with.

The test on the large graph demonstrates that the meta-algorithm may not be able to find the optimal solution without using a number of shocks that approaches infinity.

B. SECOND FORMULATION

The meta-algorithm applied to Problem 2 is much more promising. The objective value and thus the number of interface hardware sets needed varies from 82 to 96. This halves the number of interface hardware sets required over Problem 1. In addition, the solutions for the four variants are all similar. The majority of the nodes are in the same segment throughout the variants.

Further evidence of a good solution was found when the meta-algorithm was tested on the small test graph in Figure 7. It found the optimal objective value of three very quickly in all ten test runs.

For Problem 2 with K = 4 the number of interface nodes required is greatest when the third segment is being upgraded. To reduce the number of active interface nodes during this step of the upgrade, nodes are placed in the first and last segments. These "extreme" partition segments don't have an arc directly linking them to the other extreme partition. This increases the number of active interface nodes in the first partition segment, but reduces the peak size of the active sets.

When K = 5, the number of nodes in the first segment is smaller. The peak number of active interface nodes occurs when the third segment is upgraded so the algorithm attempts to relieve this pressure. The algorithm attempts to reduce the peak number by placing nodes in the extreme segments as was done when K = 4. The algorithm tends to reduce the number of nodes in the first segments that are directly connected to the segment being upgraded during the peak. This is more evident when K = 5.

Since the solutions found by the meta-algorithm are very similar and they had random starting positions, it is likely that the solutions are good. The solutions for Problem 2 for DTFIP are given in Appendix A.

VI. CONCLUSION

This thesis has developed DORS, a dynamic platform-independent solver for graph and network problems of operations research. DORS allows dynamic run-time manipulation of meta-algorithms as demonstrated by the "Add Algorithm" button and the "Objective Method" selector. DORS is extensible: The algorithms and objective-function evaluators used in DORS are independent of the solver and may be changed by adding or deleting the name of the algorithm or objective-function evaluator from a text file. Since DORS is written in Java, it is potentially platform-independent. (DORS' graphical user interface is currently tied to a vendor-specific software system that must be replaced in the future.) Since the algorithms are written in Java using Konig, they can be passively monitored. However, this capability is not currently being used in the solver.

The use of DORS is demonstrated on a graph-partitioning problem derived from a network-upgrade project of the Defense Information Systems Agency (DISA). DISA intends to upgrade a data-transmission network by partitioning the network into four or five approximately

equal-sized pieces, and then upgrading the hardware in one segment of the partition at a time. The network of concern is a portion of the Defense Information Infrastructure (DII) containing about 200 nodes and 1400 arcs.

When the hardware at a node i is upgraded and i is directly connected to another node j that is equipped only with old hardware, interface hardware (in addition to other hardware) must be installed at i. A node requiring this interface hardware is referred to as an "interface node." Two different upgrade procedures lead to two different objective functions and thus two versions of the graph-partitioning problem for DISA: Problem 1 minimizes the total number of interface nodes used throughout the upgrade process; Problem 2 minimizes the peak number of interface nodes across all steps of the upgrade process. Four variants of each problem are analyzed with DORS: The number and size of the partition segments are allowed to vary.

A library of four heuristic algorithms for graph partitioning is constructed for use in DORS. This library is then used to develop a meta-algorithm for solving the upgrade models. These algorithms are combined dynamically in a sequence of algorithms called a "meta-algorithm."

A meta-algorithm consisting of 114 invocations of the four algorithms is developed and applied to construct solutions for the DII problem. Four variations of each problem are analyzed.

Solutions for Problem 1 are of little use. Almost all of the nodes are interface nodes. The meta-algorithm is applied to a test graph of approximately the same size as DISA's network, but with a known optimal solution. The meta-algorithm cannot find a solution close to optimal indicating that the meta-algorithm is probably ineffective with this formulation.

For Problem 2, we allow the interface hardware of the interface nodes to be used multiple times. In this formulation, the partition segments are ordered from 1 to K, with the segments being upgraded in that order. At the point at which a node i in segment k is adjacent only to upgraded nodes, node i's interface hardware may be removed and used in a segment k', k' > k.

The objective is to minimize the peak number of sets of interface hardware in use at any one time.

The meta-algorithm performs much better on this formulation than it does on the first. The total number of nodes requiring interface hardware is as low as 105. All

variants of this formulation seemed to provide good solutions. The four solutions are similar and the meta-algorithm finds an optimal solution for the test problem. Solutions for this formulation, on DISA's DII problem, are provided in Appendix A for evaluation by DISA.

Despite the difficulty with Problem 1, it has been demonstrated that DORS allows for the creation of dynamic platform-independent meta-algorithms. The concept of implementing meta-algorithms to provide useful solutions is demonstrated and four possible solutions to one formulation of DISA's graph-partitioning problem are provided.

APPENDIX A. SOLUTIONS

The solutions from the application of the meta-algorithm applied to Problem 2 for DTFIP follow this page. For each solution, partition segments are listed in the proposed order of implementation. The final set of nodes marked "disconnected" have degree zero and may be upgraded at any time.

Results for K = 4, m = 40, M = 60.

The strings "D001N###" are node identifiers. The number following the node identifiers indicates for which steps the node needs active interface hardware. The number required for each step, in order, are 56, 85, 87.

Segment 1
D001N008 123
D001N009 123
D001N026 123
D001N027 123
D001N028 123
D001N029 12
D001N030 12
D001N032 123
D001N037 123
D001N038 123
D001N041 1
D001N043 123
D001N045 123
D001N046 123
D001N052 123
D001N064 123
D001N066 123
D001N072 123
D001N080 123
D001N081 123
D001N082 12
D001N086 123
D001N088 123
D001N089 123
D001N090 123
D001N102 123
D001N105 12
D001N110 1
D001N120 123
D001N121 123
D001N136 123
D001N137 123
D001N148 12
D001N160 12
D001N162 123
D001N163 123
D001N165 123
D001N175 123
D001N181 123
D001N182 123
D001N184 123
D001N188 123
D001N192 123
D001N198 123
D001N202 12
D001N204 123
D001N209
D001N214 123
D001N215 123
D001N221 123
D001N225 123
D001N227 123
D001N234 123
D001N237 123
D001N241 123
D001N246 123
D001N248 123

Segment 2
D001N018 23
D001N019
D001N033
D001N050 23
D001N051 3
D001N055 23
D001N056 23
D001N061 23
D001N065 23
D001N071 23
D001N076 3
D001N078
D001N079 23
D001N091 23
D001N094 23
D001N096 23
D001N098 23
D001N099 23
D001N111
D001N117 23
D001N124 23
D001N126
D001N127 23
D001N142 23
D001N145 23
D001N164
D001N166 23
D001N168 23
D001N169 23
D001N176 23
D001N186 23
D001N187 23
D001N189 23
D001N201 23
D001N207 23
D001N211 23
D001N218
D001N224 23
D001N238 23
D001N245 23

Segment 3
D001N004
D001N006
D001N013 3
D001N015
D001N017
D001N021
D001N031
D001N035
D001N039
D001N053 3
D001N070 3
D001N074
D001N075
D001N077
D001N085
D001N087
D001N093
D001N095
D001N100
D001N101 3
D001N103 3
D001N104
D001N109
D001N112
D001N115 3
D001N122
D001N125
D001N129
D001N132
D001N133 3

D001N141
D001N143
D001N149
D001N151
D001N159
D001N172
D001N183
D001N185
D001N191
D001N194
D001N200
D001N206
D001N210
D001N212
D001N223
D001N228
D001N235
D001N239
D001N244
D001N250

Segment 4
D001N001
D001N003
D001N007
D001N014
D001N016
D001N020
D001N023
D001N024
D001N025
D001N036
D001N040
D001N042
D001N048
D001N057
D001N059
D001N063
D001N067
D001N069
D001N084
D001N097
D001N107
D001N108
D001N113
D001N114
D001N116
D001N123
D001N130
D001N139
D001N144
D001N146
D001N147
D001N150
D001N152
D001N153
D001N156
D001N157
D001N158
D001N161
D001N167
D001N171
D001N174
D001N179
D001N180
D001N190
D001N203
D001N208
D001N213
D001N216
D001N217
D001N219
D001N220
D001N222
D001N226
D001N229
D001N233
D001N236
D001N240
D001N243
D001N247
D001N249

Disconnected
D001N002
D001N060
D001N119
D001N134
D001N138
D001N140
D001N196
D001N230
D001N231
D001N232

Results for K = 4, m = 35, M = 65.

The strings "D001N###" are node identifiers. The number following the node identifiers indicates for which steps the node needs active interface hardware. The number required for each step, in order, are 52, 78, 84.

Segment 1
D001N008 123
D001N009 123
D001N026 123
D001N027 123
D001N028 123
D001N029 12
D001N030 12
D001N032 123
D001N037 123
D001N038 123
D001N041 1
D001N043 123
D001N045 123
D001N046 123
D001N052 123
D001N064 123
D001N066 123
D001N072 123
D001N080 123
D001N081 123
D001N082 12
D001N086 123
D001N088 123
D001N089 123
D001N090 123
D001N102 123
D001N105 12
D001N110 123
D001N120 123
D001N121 123
D001N148 12
D001N160 12
D001N162 123
D001N163 123
D001N165 123
D001N181 123
D001N182 123
D001N188 123
D001N192 123
D001N198 123
D001N202 12
D001N204 123
D001N209 123
D001N214 123
D001N215 12
D001N225 12
D001N227 123
D001N234 123
D001N237 123
D001N241 123
D001N246 12
D001N248 123

Segment 2
D001N018 3
D001N019
D001N033
D001N050 23
D001N051 23
D001N055 3
D001N056 23
D001N061 23
D001N065 23
D001N071 23
D001N076 23
D001N078
D001N079 3
D001N091 23
D001N094 23
D001N096 23
D001N098 23
D001N099 23
D001N111
D001N117 3
D001N124 23
D001N126
D001N127 23
D001N142 3
D001N145 23
D001N164
D001N166 23
D001N168 23
D001N169 23
D001N176 23
D001N186 23
D001N187 23
D001N189 23
D001N201 3
D001N207 23
D001N211 23
D001N218
D001N224 23
D001N238 23
D001N245 23

Segment 3
D001N004
D001N006
D001N013 3
D001N015
D001N017
D001N021 3
D001N031
D001N035
D001N039
D001N053 3
D001N070 3
D001N074
D001N075
D001N077
D001N085
D001N087
D001N093
D001N095
D001N100
D001N101 3
D001N103 3
D001N104
D001N109
D001N112
D001N115 3
D001N122
D001N125
D001N129
D001N132
D001N133 3
D001N141
D001N143
D001N149
D001N151 3
D001N159
D001N172
D001N183
D001N185 3

D001N191
D001N194
D001N200
D001N206
D001N210
D001N212
D001N223
D001N228
D001N235
D001N239
D001N244
D001N250

Segment 4
D001N001
D001N003
D001N007
D001N014
D001N016
D001N020
D001N023
D001N024
D001N025
D001N036
D001N040
D001N042
D001N048
D001N057
D001N059
D001N063
D001N067
D001N069
D001N084
D001N097
D001N107
D001N108
D001N113
D001N114
D001N116
D001N123
D001N130
D001N136
D001N137
D001N139
D001N144
D001N146
D001N147
D001N150
D001N152
D001N153
D001N156
D001N157
D001N158
D001N161
D001N167
D001N171
D001N174
D001N175
D001N179
D001N180
D001N184
D001N190
D001N203
D001N208
D001N213
D001N216
D001N217
D001N219
D001N220
D001N221
D001N222
D001N226
D001N229
D001N233
D001N236
D001N240
D001N243
D001N247
D001N249

Disconnected
D001N006
D001N060
D001N119
D001N134
D001N138
D001N140
D001N196
D001N230
D001N231
D001N232

Results for K = 5, m = 30, M = 50.

The strings "D001N###" are node identifiers. The number following the node identifiers indicates for which steps the node needs active interface hardware. The number required for each step, in order, are 33, 72, 96, 85.

Segment 1
D001N006 1234
D001N008 1234
D001N015 1234
D001N021 1234
D001N028 123
D001N046 1234
D001N063 1234
D001N074 1234
D001N079 1234
D001N081 1234
D001N088 1234
D001N101 1234
D001N110 123
D001N111 1234
D001N115
D001N116 1
D001N126 1234
D001N141 1234
D001N146 1234
D001N147 1234
D001N149 1234
D001N150 1234
D001N151 1234
D001N156 1234
D001N161 1234
D001N162 1234
D001N164 1234
D001N179 1234
D001N182 123
D001N187 1234
D001N188 1234
D001N190 1234
D001N211 12
D001N245 1234
D001N250

Segment 2
D001N001 234
D001N031 234
D001N032 234
D001N038 234
D001N042 234
D001N052 234
D001N053 234
D001N055 234
D001N059 234
D001N065 234
D001N069 234
D001N077 234
D001N078
D001N082 234
D001N085 234
D001N087 23
D001N090 234
D001N094 23
D001N096 234
D001N102 23
D001N105 23
D001N117 23
D001N120 234
D001N122 234
D001N130 23
D001N132 234
D001N139
D001N148 234
D001N152 234
D001N166 234
D001N189 234
D001N201 234
D001N208 234
D001N215 23
D001N217 234
D001N224 234
D001N227 23
D001N229 234
D001N233 234
D001N236 23
D001N239 234
D001N246 234

Segment 3
D001N016 34
D001N017 34
D001N039 3
D001N064 34
D001N066 34
D001N089 34
D001N093 3
D001N095 3
D001N103 3
D001N109 3
D001N112 34
D001N113
D001N114
D001N127 3
D001N142 34
D001N143
D001N153 34
D001N171 34
D001N180 34
D001N186
D001N198 34
D001N210 34
D001N214 34
D001N222 34
D001N228 34
D001N234 3
D001N235 34
D001N238
D001N243 34
D001N248 34

Segment 4
D001N003
D001N007
D001N013
D001N014
D001N018 4
D001N020
D001N026
D001N027
D001N030
D001N033 4
D001N035
D001N036
D001N041
D001N048
D001N050
D001N057 4
D001N061
D001N075
D001N097
D001N100
D001N107

D001N125
D001N129
D001N137
D001N144
D001N145 4
D001N157
D001N160
D001N165
D001N167
D001N169
D001N172
D001N174 4
D001N175 4
D001N184
D001N185
D001N192
D001N204
D001N206
D001N212
D001N216
D001N218 4
D001N219 4
D001N221
D001N225
D001N226
D001N241
D001N244
D001N247
D001N249

Segment 5
D001N004
D001N009
D001N019
D001N023
D001N024
D001N025
D001N029
D001N037
D001N040
D001N043
D001N045
D001N051
D001N056
D001N067
D001N070
D001N071
D001N072
D001N076
D001N080
D001N084
D001N086
D001N091
D001N098
D001N099
D001N104
D001N108
D001N121
D001N123
D001N124
D001N133
D001N136
D001N158
D001N159
D001N163
D001N168
D001N176
D001N181
D001N183
D001N191
D001N194
D001N200
D001N202
D001N203
D001N207
D001N209
D001N213
D001N220
D001N223
D001N237
D001N240

Disconnected
D001N002
D001N060
D001N119
D001N134
D001N138
D001N140
D001N196
D001N230
D001N231
D001N232

Results for K = 5, m = 25, M = 55.

The strings "D001N###" are node identifiers. The number following the node identifiers indicates for which steps the node needs active interface hardware. The number required for each step, in order, are 31, 65, 82, 78.

Segment 1
D001N006 1234
D001N008 1234
D001N021 1234
D001N028 123
D001N046 1234
D001N063 1234
D001N074 1234
D001N079 1234
D001N081 1234
D001N088 1234
D001N101 1234
D001N110 12
D001N111 1234
D001N115
D001N116 1
D001N126 1234
D001N141 1234
D001N146 1234
D001N147 1234
D001N149 1234
D001N150 1234
D001N151 1234
D001N156 1234
D001N161 1234
D001N162 1234
D001N179 1234
D001N182 123
D001N187 1234
D001N188 1234
D001N190 1234
D001N211 123
D001N245 1234
D001N250

Segment 2
D001N001 234
D001N031 234
D001N032 234
D001N038 234
D001N042 234
D001N052 234
D001N053 234
D001N055 234
D001N059 234
D001N065 234
D001N069 234
D001N077 234
D001N078
D001N082 234
D001N085 234
D001N087 234
D001N090 234
D001N094 2
D001N096 234
D001N102 23
D001N117 234
D001N120 234
D001N122 234
D001N132 234
D001N139
D001N148 234
D001N189 234
D001N201 234
D001N208 234
D001N217 234
D001N224 234
D001N227 234
D001N229 234
D001N233 234
D001N236 2
D001N239 234
D001N246 234

Segment 3
D001N016 34
D001N017 34
D001N039 34
D001N064 34
D001N066 34
D001N093 34
D001N095 34
D001N103
D001N109
D001N113
D001N114
D001N127
D001N142 34
D001N143
D001N153 34
D001N171 34
D001N180 34
D001N198 34
D001N210 34
D001N214 34
D001N222 34
D001N228 34
D001N234 34
D001N235 34
D001N238
D001N243 34
D001N248 34

Segment 4
D001N003
D001N007
D001N013
D001N014
D001N015
D001N018
D001N020
D001N026
D001N027
D001N030
D001N033
D001N035
D001N036
D001N041
D001N048
D001N050
D001N057
D001N061
D001N075
D001N089 4
D001N097
D001N100
D001N105
D001N107
D001N112 4
D001N125
D001N129
D001N137
D001N144
D001N145
D001N152 4

D001N157
D001N160
D001N164 4
D001N165
D001N166 4
D001N167
D001N169
D001N172
D001N174
D001N175
D001N184
D001N185
D001N192
D001N204
D001N206
D001N212
D001N215 4
D001N216
D001N218
D001N219
D001N221
D001N225
D001N226
D001N241
D001N244
D001N247
D001N249

Segment 5
D001N004
D001N009
D001N019
D001N023
D001N024
D001N025
D001N029
D001N037
D001N040
D001N043
D001N045
D001N051
D001N056
D001N067
D001N070
D001N071
D001N072
D001N076
D001N080
D001N084
D001N086
D001N091
D001N098
D001N099
D001N104
D001N108
D001N121
D001N123
D001N124
D001N130
D001N133
D001N136
D001N158
D001N159
D001N163
D001N168
D001N176
D001N181
D001N183
D001N186
D001N191
D001N194
D001N200
D001N202
D001N203
D001N207
D001N209
D001N213
D001N220
D001N223
D001N237
D001N240

Disconnected
D001N002
D001N060
D001N119
D001N134
D001N138
D001N140
D001N196
D001N230
D001N231
D001N232

APPENDIX B. META-ALGORITHM RUNS

DORS is executed on a 266 MHz Pentium II processor under Windows NT 4.0. The same sequence of 114 invocations of the four algorithms presented in this thesis is used in the formation of the meta-algorithm used to solve the problems defined in Chapter III. However, the number of partitions, minimum partition size, maximum partition size, and the objective function are varied with each run of the meta-algorithm.

The meta-algorithm starts with five random solutions provided by GetInitialSolution, each followed by Change1Node for refinement. This provides five separate solutions: The best solution is kept and the other four are discarded by the meta-algorithm. Next, Swap2Nodes is used on the best solution found so far. After Swap2Nodes, 51 repetitions of RandomizeXNodes, each followed by Change1Node, are run. This is done to shock the system as was described in Chapter III. The first shock changes 100 nodes randomly and each subsequent shock reduces that number by one until the last shock changes only 50 nodes. After the final shock is complete and Change1Node has found

a local optimum, the best solution found to this point is sent to Swap2nodes and the meta-algorithm concludes.

DORS is designed using Borland's JBuilder©, which has a very slow Java virtual machine. This causes the meta-algorithm to run extremely sluggishly, making execution times far too long. To reduce this time, the solver needs to be executed separately from JBuilder with a state-of-the-art, just-in-time Java virtual machine. Symantec's© Java system was acquired through the Internet and used for this purpose.

Unfortunately DORS uses JBuilder's interface builder to place objects in the GUI, and this makes the DORS design fall short of 100 percent Java compatibility. Thus, it is not possible to execute DORS with Symantec's Java virtual machine.

To calculate the approximate time the solver would take with a state-of-the-art Java virtual machine, the execution time for Problem 1 with M= 60, m = 40, and K = 4 is used as a baseline. A revision of DORS with the meta-algorithm hardwired and with the GUI disabled is executed. The execution time is 55 seconds compared to an execution time of 6456 seconds with DORS under JBuilder. Therefore,

to reflect the computational times probably achievable using a state-of-the-art Java virtual machine, all execution times recorded by DORS are divided by a factor of $117 \approx 6456/55$.

The cumulative execution time and objective value for several points in the meta-algorithm for all runs of the meta-algorithm follow.

Algorithm number	Selected step in algorithm	Adjusted Cumulative Seconds	Objective Value
1	GetInitialSolution (1)	<1	206
2	Change1Node (1)	<1	201
3	GetInitialSolution (2)	<1	204
5	Change1Node (2)	<1	199
7	Change1Node (3)	<1	190
9	Change1Node (4)	<1	196
11	Change1Node (5)	<1	198
12	Swap2Nodes (1)	12	190
13	RandomizeXNodes (First Instance)	12	190
33	RandomizeXNodes (Eleventh Instance)	23	190
53	RandomizeXNodes (Twenty-first Instance)	30	190
73	RandomizeXNodes (Thirty-first Instance)	37	190
93	RandomizeXNodes (Forty-first Instance)	45	190
114	Swap2Nodes (2)	55	190

Table 2. Objective function values and adjusted cumulative time for selected steps of the meta-algorithm for Problem 1, m = 40 , M = 60, K = 4.

Algorithm number	Selected step in algorithm	Adjusted Cumulative Seconds	Objective Value
1	GetInitialSolution (1)	<1	204
2	Change1Node (1)	<1	194
3	GetInitialSolution (2)	<1	202
5	Change1Node (2)	<1	200
7	Change1Node (3)	<1	194
9	Change1Node (4)	<1	194
11	Change1Node (5)	<1	195
12	Swap2Nodes (1)	11	195
13	RandomizeXNodes (First Instance)	11	190
33	RandomizeXNodes (Eleventh Instance)	22	190
53	RandomizeXNodes (Twenty-first Instance)	28	190
73	RandomizeXNodes (Thirty-first Instance)	36	190
93	RandomizeXNodes (Forty-first Instance)	43	190
114	Swap2Nodes (2)	53	190

Table 3. Objective function values and adjusted cumulative time for selected steps of the meta-algorithm for Problem 1, m = 35, M = 65, K = 4.

Algorithm number	Selected step in algorithm	Adjusted Cumulative Seconds	Objective Value
1	GetInitialSolution (1)	<1	205
2	Change1Node (1)	<1	198
3	GetInitialSolution (2)	<1	205
5	Change1Node (2)	<1	199
7	Change1Node (3)	<1	201
9	Change1Node (4)	<1	197
11	Change1Node (5)	<1	199
12	Swap2Nodes (1)	12	199
13	RandomizeXNodes (First Instance)	12	199
33	RandomizeXNodes (Eleventh Instance)	21	196
53	RandomizeXNodes (Twenty-first Instance)	28	196
73	RandomizeXNodes (Thirty-first Instance)	35	196
93	RandomizeXNodes (Forty-first Instance)	43	196
114	Swap2Nodes (2)	52	196

Table 4. Objective function values and adjusted cumulative time for selected steps of the meta-algorithm for Problem 1, m = 30, M = 50, K = 5.

Algorithm number	Selected step in algorithm	Adjusted Cumulative Seconds	Objective Value
1	GetInitialSolution (1)	<1	204
2	Change1Node (1)	<1	190
3	GetInitialSolution (2)	<1	205
5	Change1Node (2)	<1	192
7	Change1Node (3)	<1	196
9	Change1Node (4)	<1	200
11	Change1Node (5)	<1	193
12	Swap2Nodes (1)	12	190
13	RandomizeXNodes (First Instance)	12	190
33	RandomizeXNodes (Eleventh Instance)	21	190
53	RandomizeXNodes (Twenty-first Instance)	28	190
73	RandomizeXNodes (Thirty-first Instance)	35	188
93	RandomizeXNodes (Forty-first Instance)	43	188
114	Swap2Nodes (2)	52	188

Table 5. Objective function values and adjusted cumulative time for selected steps of the meta-algorithm for Problem 1, m = 25, M = 55, K = 5.

Algorithm number	Selected step in algorithm	Adjusted Cumulative Seconds	Objective Value
1	GetInitialSolution (1)	<1	144
2	Change1Node (1)	<1	114
3	GetInitialSolution (2)	<1	132
5	Change1Node (2)	<1	117
7	Change1Node (3)	<1	108
9	Change1Node (4)	<1	111
11	Change1Node (5)	<1	103
12	Swap2Nodes (1)	31	95
13	RandomizeXNodes (First Instance)	31	94
33	RandomizeXNodes (Eleventh Instance)	38	91
53	RandomizeXNodes (Twenty-first Instance)	42	91
73	RandomizeXNodes (Thirty-first Instance)	51	88
93	RandomizeXNodes (Forty-first Instance)	58	87
114	Swap2Nodes (2)	70	87

Table 6. Objective function values and adjusted cumulative time for selected steps of the meta-algorithm for Problem 2, m = 40, M = 60, K = 4.

Algorithm number	Selected step in algorithm	Adjusted Cumulative Seconds	Objective Value
1	GetInitialSolution (1)	<1	146
2	Change1Node (1)	<1	111
3	GetInitialSolution (2)	<1	132
5	Change1Node (2)	<1	114
7	Change1Node (3)	<1	108
9	Change1Node (4)	<1	111
11	Change1Node (5)	<1	90
12	Swap2Nodes (1)	27	88
13	RandomizeXNodes (First Instance)	27	85
33	RandomizeXNodes (Eleventh Instance)	34	85
53	RandomizeXNodes (Twenty-first Instance)	42	84
73	RandomizeXNodes (Thirty-first Instance)	49	84
93	RandomizeXNodes (Forty-first Instance)	56	84
114	Swap2Nodes (2)	71	84

Table 7. Objective function values and adjusted cumulative time for selected steps of the meta-algorithm for Problem 2, m = 35, M = 65, K = 4.

Algorithm number	Selected step in algorithm	Adjusted Cumulative Seconds	Objective Value
1	GetInitialSolution (1)	<1	123
2	Change1Node (1)	<1	114
3	GetInitialSolution (2)	<1	129
5	Change1Node (2)	<1	117
7	Change1Node (3)	<1	100
9	Change1Node (4)	<1	111
11	Change1Node (5)	<1	105
12	Swap2Nodes (1)	34	97
13	RandomizeXNodes (First Instance)	34	97
33	RandomizeXNodes (Eleventh Instance)	39	97
53	RandomizeXNodes (Twenty-first Instance)	46	97
73	RandomizeXNodes (Thirty-first Instance)	52	96
93	RandomizeXNodes (Forty-first Instance)	59	96
114	Swap2Nodes (2)	82	96

Table 8. Objective function values and adjusted cumulative time for selected steps of the meta-algorithm for Problem 2, m = 30, M = 50, K = 5.

Algorithm number	Selected step in algorithm	Adjusted Cumulative Seconds	Objective Value
1	GetInitialSolution (1)	<1	121
2	Change1Node (1)	<1	90
3	GetInitialSolution (2)	<1	124
5	Change1Node (2)	<1	85
7	Change1Node (3)	<1	103
9	Change1Node (4)	<1	89
11	Change1Node (5)	<1	97
12	Swap2Nodes (1)	41	93
13	RandomizeXNodes (First Instance)	41	91
33	RandomizeXNodes (Eleventh Instance)	47	84
53	RandomizeXNodes (Twenty-first Instance)	53	84
73	RandomizeXNodes (Thirty-first Instance)	59	82
93	RandomizeXNodes (Forty-first Instance)	65	82
114	Swap2Nodes (2)	74	82

Table 9. Objective function values and adjusted cumulative time for selected steps of the meta-algorithm for Problem 2, m = 25, M = 55, K = 5.

LIST OF REFERENCES

[1] Pothen, H. Simon, and K. Liou, "Partitioning Sparse Matrices with Eigenvectors of Graphs," *Matrix Analysis Applications*, 11, 1990, pp. 430-452.

[2] B. Mohar, "The Laplacian Spectrum of Graphs," *6th International Conference on Theory and Application of Graphs*, John Wiley, New York, 1988, pp. 871-898.

[3] B. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Systems Technical Journal*, 29, 1970, pp. 291-307.

[4] Defense Information Systems Agency, "Mission and Mandate," *http://www.disa.mil/missman.html.*

[5] Chairman of the Joint Chiefs of Staff, Joint Vision 2010: America's Military: Preparing for Tomorrow, Joint Chiefs of Staff, Washington D.C., 1996.

[6] University of Oklahoma, College of Business Administration. "ATM: Asynchronous Transmission Mode," *http://www.busn.ucok.edu/tips/info_hrd/ATM.HTM.*

[7] University of Oklahoma, College of Business Administration. "Statistical Multiplexing," *http://www.busn.ucok.edu/tips/info_hrd/S_MUX.HTM.*

[8] H. Simon, "Partitioning of Unstructured Mesh Problems for Parallel Processing," *Computer Systems Engineering*, 2, 1991, pp. 135-148.

[9] M. Bazaraa, H. Sherali, and C. Shetty, Nonlinear Programming Theory and Algorithms, 2nd ed., John Wiley and Sons, Inc., New York, 1993.

[10] G. Fox and W. Furmanski, "Load Balancing Loosely Synchronous Problems with a Neural Network," http://www.npac.syr.edu/.

[11] D. Ackley, M. Littman, "A Case for Lamarckian Evolution," *Proceedings of the Workshop on Artificial Life*, 1992, pp. 3-10.

[12] A. Dekkers and E. Aarts, "Global Optimization and Simulated Annealing," *Mathematical Programming*, 50, pp. 367-393, 1991.

[13] G. Cornell and C. Hortsmann, Core Java Volume 1.1 Volume I-Fundamentals, Mountain View, CA, Prentice Hall, 1997.

[14] T. Cormen, C. Leiserson, and R. Rivest, Introduction to Algorithms, The MIT Press, Cambridge, MA., 1990.

[15] E. Aarts, Simulated Annealing and Boltzmann Machines, A Stochastic Approach to Combinatorial Optimization and Neural Networks, John Wiley and Sons Ltd, New York, 1989.

[16] L. Jackson, "Konig Beta 1.0 User Guide," http://www.trac.nps.navy.mil/jacksonl/konig/.

[17] H. Simon, "The Partitioning Problem," http://www.nas.nasa.gov/Pubs/TechReports/RNRreports/hsimon/RNR-91-008/Section3_1.html.

[18] J. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, MI, 1975.

[19] Borland JBuilder, Version 2.0, Inprise Corporation, 1998.

INITIAL DISTRIBUTION LIST

No. Copies

1. Defense Technical Information Center.....2
   8725 John J. Kingman Rd., Ste 0944
   Ft Belvoir, VA 22060-6218

2. Dudley Knox Library.....2
   Naval Postgraduate School
   411 Dyer Road
   Monterey, CA 93943-5101

3. Professor Gordon Bradley, Code OR/BZ.....4
   Department of Operations Research
   Naval Postgraduate School
   Monterey, CA 93943-5000

4. Professor R. K. Wood, Code OR/Wd.....4
   Department of Operations Research
   Naval Postgraduate School
   Monterey, CA 93943-5000

5. LT V. S. Schwartz 2
   2503 Broadmoor
   Bryan, TX 77802

6. MAJOR Leroy A. Jackson.....1
   U.S. Army TRADOC Analysis Center
   Monterey, CA 93943-5000